Secure, Expressive, and Debuggable Large-Scale Analytics

Ankur Dave

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-143
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-143.html

August 12, 2020

Secure, Expressive, and Debuggable Large-Scale Analytics

by

Ankur Dave

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Raluca Ada Popa
Professor John Chuang

Summer 2020

Secure, Expressive, and Debuggable Large-Scale Analytics

Abstract

Secure, Expressive, and Debuggable Large-Scale Analytics

by

Ankur Dave

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Growing volumes of data collection, outsourced computing, and demand for complex analytics have led to the rise of big data analytics frameworks such as MapReduce and Apache Spark. However, these systems fall short in processing sensitive data, graph querying, and debugging. This dissertation addresses these remaining challenges in analytics by introducing three systems built on top of Spark: Oblivious Coopetitive Queries (OCQ), GraphFrames, and Arthur. OCQ focuses on the setting of coopetitive analytics, which refers to cooperation among competing parties to run queries over their joint data. OCQ is an efficient, general framework for oblivious coopetitive analytics using hardware enclaves. GraphFrames is an integrated system that lets users combine graph algorithms, pattern matching, and relational queries, each of which typically requires a specialized engine, and optimizes work across them. Arthur is a debugger for Apache Spark that provides a rich set of analysis tools at close to zero runtime overhead through selective replay of data flow applications. Together, these systems bring Apache Spark closer to the goal of a unified analytics platform that retains the flexibility, extensibility, and performance of relational systems.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I am thankful to my advisors Ion Stoica and Raluca Ada Popa. This dissertation would not have been possible without their wisdom, patience, and trust.

I was lucky to work with Matei Zaharia and Scott Shenker starting in my undergraduate years. I learned most of my research skills from Matei and he inspired me to pursue graduate school. I admired Scott's approach to research and teaching and he ultimately convinced me to come to Berkeley.

In my early years as a graduate student I worked with Joseph E. Gonzalez and Reynold Xin. Joey taught me valuable lessons about research and I am grateful for his encouragement over the years. Reynold is responsible for my enduring interest in databases, particularly efficient query processing.

I worked closely with Wenting Zheng on Opaque [145]. Our complementary skills and personalities made this collaboration a highlight of my time at Berkeley.

My brother Arjun and his partner Adora have been with me throughout graduate school and I am a better person thanks to them. Finally, I thank my parents Rashmi and Salil for their incredible love and support.

Chapter 1

Introduction

Computer science historically focused on machines, algorithms, and data structures: specific methods that work together to transform data into insights. Programming languages abstracted away specifics of different machines. The insight of databases was to further abstract away specifics of data representation and algorithms, and expose a simpler model of data as a collection of relations that can be transformed using queries expressed using a small set of relational operators. This gives the database the flexibility to choose which algorithms and data structures are best suited for the task.

In real use, databases came to take on two different roles: as the data store of record, and as an environment for extracting insights from that data. Systems began to specialize for one or the other. This dissertation is concerned with the latter role, which has come to be known as analytics.

1.1 Rise of Distributed Dataflow Systems

Analytics has traditionally been performed using specialized data warehouses based on the Massively Parallel Processing (MPP) architecture, in which data is extracted, transformed, and loaded into a shared-nothing database [47] that stores data in columnar format [126] and supports efficient SQL query execution through vectorization [20] or code generation [104]. Data warehouses often use a cluster composed of a small number of high-end physical machines.

Since the data warehouse became popular, a few identifiable shifts have occurred in the area of analytical databases:

1. *Growing volumes of data collection*, driven by globalization, the internet, IoT, and the outsized role of software across all industrial and government sectors. This motivated the creation of the MapReduce [44] framework, which scaled beyond the limits of a traditional data warehouse using hundreds or thousands of commodity machines.

2. *Outsourced computing*. Rather than each organization needing to maintain its own cluster, it has become more economical and convenient to outsource analytics to a small number of cloud providers. This offers a number of advantages including elasticity, ease of management,

and improved resource utilization through disaggregation of compute and storage resources. It has also created new security and regulatory concerns in case of a breach.

3. *Demand for more complex analytics*. In addition to relational operations on tabular data, it is increasingly important to view data as graphs or to apply machine learning techniques. These operations are cumbersome to express in SQL, even with user-defined functions (UDFs), creating demand for more specialized frameworks [89, 137, 91, 59]. The flexibility offered by these frameworks in turn makes these operations more difficult to debug than typical SQL queries.

These shifts led to the rise of distributed dataflow systems based on the MapReduce framework, including Dryad [72], Apache Spark [142], and Apache Flink [29] for general data processing; and Dremel [96], Apache Drill [5], Apache Impala [80], Presto [115], and Apache Spark SQL [11] for SQL analytics. These systems support unstructured data in addition to tabular data, and they execute data-parallel operations by breaking them into directed acyclic graphs (DAGs) of deterministic tasks that each operate on a horizontal partition of the input data and can be scheduled in multiple locations and re-executed in case of failure. Their architecture emphasizes extreme scalability. Loosely coupled internals make them adaptable to different storage services and applications. They offer fine-grained, parallel fault recovery, making them suitable for executing long-running, complex queries on cloud machines that can scale elastically and can be treated as interchangeable.

1.2 Challenges in Large-Scale Analytics

Although distributed dataflow systems have seen tremendous growth, we argue that they have not fully addressed these challenges, posing a barrier to further adoption. They do not address the following needs, illustrated in Figure 1.1.

1. *Processing sensitive federated data*. Tools for large-scale analytics are designed to exploit the elasticity and convenient disaggregated services offered by cloud providers. However, many organizations are hesitant to entrust their most sensitive data to cloud providers, and regulations may prevent them from doing so [30]. In addition, these tools are designed to be operated by a single organization and cannot securely analyze data from multiple untrusting organizations. Traditional federated databases [127, 144, 53, 52] assume participating organizations trust each other and can share data. These systems are unsuitable when organizations are in business competition or are legally prevented from sharing data. For example, banks may wish to collaboratively assess industrywide risk, but they cannot share their proprietary customer information with one another in plaintext. Existing systems in this setting of *coopetitive* analytics (analytics among cooperative and competing parties) either assume a weak semi-honest threat model or incur prohibitive overhead [134, 136, 15, 103].

2. *Graph queries.* Specialized graph mining tools support efficient pattern querying [137, 132, 130], but these systems have limited support for other graph analytics tasks such as distributed

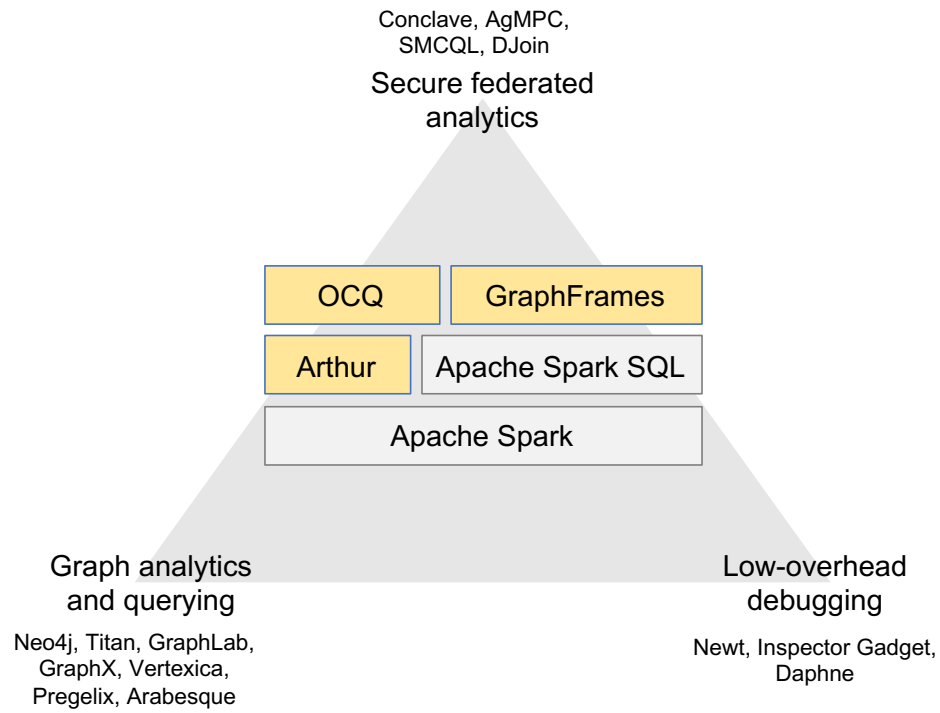


Figure 1.1: No existing system for large-scale analytics addresses all three identified needs, represented as the vertices of the triangle in this figure. This dissertation introduces three systems built on Apache Spark and Apache Spark SQL to address these needs.

graph algorithms. A workflow involving multiple systems is cumbersome to orchestrate and incurs significant overhead when data crosses system boundaries. Naively encoding graph queries as join-heavy dataflow computation results in very poor performance due to the large intermediate result sizes generated. Prior work [59, 75, 25, 50] has explored how to express iterative graph algorithms as dataflow computation, but these systems either do not support graph queries, are unable to share intermediate results across multiple queries in a workload, or forego intra-query fault tolerance.

3. *Debugging.* It is difficult to understand the correctness and performance of complex analytics queries due to their large scale and the scarcity of tools that go beyond coarse metrics and error reporting without incurring significant overhead. Tools for testing assertions, tracing through data flows, and replaying code exist [109, 73, 58, 43], but they are too expensive to use in production.

This dissertation addresses these three challenges in large-scale analytics, toward the goal of a unified analytics platform that retains the flexibility, extensibility, and performance of relational systems.

1.3 Dissertation Overview

The work in this dissertation applies generally to any distributed dataflow system. For concreteness, we choose to build on Apache Spark [142], a popular distributed dataflow framework, and Apache Spark SQL [11], a distributed SQL analytics engine that uses Spark. This dissertation presents three systems that build on Apache Spark and Apache Spark SQL to address the aforementioned challenges in large-scale analytics.

1. *OCQ* (Chapter 2) enables secure analytics in a multi-party setting. OCQ extends Spark SQL to execute multi-party queries securely using hardware enclaves. Its query planner chooses how and where to execute each relational operator to prevent data leakage through side channels such as memory access patterns, network traffic statistics, and cardinality, while minimizing overhead. We find that OCQ is up to 9.9x faster than Opaque, a state-of-the-art secure analytics framework which outsources all data and computation to an enclave-enabled cloud; and is up to 219x faster than implementing analytics using AgMPC, a state-of-the-art secure multi-party computation framework.

   This chapter is adapted from previously published work [42].

2. *GraphFrames* (Chapter 3) enables users to express graph algorithms, pattern matching and relational queries, and optimizes work across them. It executes these operations efficiently by materializing multiple views of the graph, selecting join plans based on the available views, and executing these plans using Spark SQL.

   This chapter is an extended version of previously published work [41].

3. *Arthur* (Chapter 4) is a debugger for Spark programs that provides a rich set of analysis tools at close to zero runtime overhead through selective replay. Unlike previous replay debuggers, which add high overheads due to the need to log low-level nondeterministic events, Arthur takes advantage of the structure of Spark programs, which are composed of graphs of deterministic tasks for fault tolerance, to minimize its logging cost. It uses selective replay to implement a variety of debugging features, including rerunning any task in a single-process debugger; ad-hoc queries on computation state; and forward and backward tracing of records through the computation, which it achieves using a program transformation at replay time.

Finally, Chapter 5 summarizes the main results and discusses directions for future work.

Chapter 2

Oblivious Coopetitive Analytics Using Hardware Enclaves

In this chapter we address challenges created by the shift toward outsourced and multi-party analytics. This chapter builds upon Opaque [145], a system I coauthored that enables large-scale analytics on sensitive data. Opaque allows a single organization to outsource its computation to an untrusted cloud. In this chapter we study the more general problem of how to securely process data from multiple untrusting organizations.

2.1 Introduction

Distributed analytics frameworks [142] are now widely used, but are designed to operate on data owned by one entity. Federated databases, which span data owned by multiple cooperating parties, have a long history in the database community [127, 144, 53, 52]. This community has focused on the case when the organizations trust each other and can share data.

However, there are many applications in which organizations *cannot share plaintext data* with

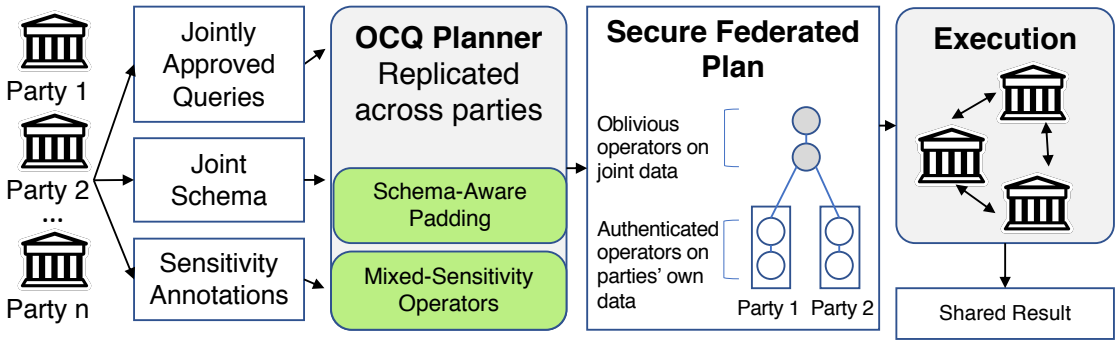


Figure 2.1: OCQ executes approved federated queries using hardware enclaves. Its query planner and relational operators hide memory and network access patterns and sensitive cardinalities.

each other because, for example, they are in business competition, or due to privacy regulations and liability concerns. Nevertheless, collaboration among these competing organizations could enable new applications. For example, banks would like to perform analytics over their aggregate data to better detect money laundering, but cannot share the data with each other because they are in competition. The Chief Risk Officer of Scotiabank stated that such collaboration "will enhance yields by orders of magnitude" [48]. We are partnering with Scotiabank for this use case. As another example, consider a consortium of hospitals that want to pool their patient data for researchers to access. Researchers need the ability to correlate patients across hospitals, yet regulations prevent the hospitals from sharing raw patient data or letting it leave the premises [30]. Following previous work [146], we refer to this setting of analytics among cooperative and competing parties as *coopetitive* analytics.

In a coopetitive setting, parties agree in advance on a shared schema and a set of allowed queries. They agree to run these queries on their private data and share only the results. In particular, they will not share the actual database of any party or intermediate results in the query computation.

Prior work in the coopetitive setting uses either specialized cryptography or hardware enclaves. Cryptography [114, 15, 146, 2, 79, 95] either offers limited functionality too restrictive for general analytics, as is the case for partially-homomorphic encryption, or introduces enormous overheads, taking hours to months for typical queries, as is the case for secure multi-party computation (MPC), as we show in §2.8.

Approaches based on hardware enclaves [145, 49, 98, 107] are more promising for performance. While hardware enclaves have many side channels [135, 24, 123, 139, 26, 84], prior work [68] shows that many classes of side channels disappear if one designs the computation to be *oblivious*: memory accesses are independent of sensitive data. In a distributed setting, network traffic patterns also create a side channel [106]. Hence, to use hardware enclaves for coopetitive analytics requires oblivious protocols designed for this setting. The overarching challenge is that such protocols are slow.

The most relevant related work to OCQ is likely Opaque [145], which offers oblivious analytics on data outsourced to a *single* untrusted cloud. However, aggregating multiple parties' sensitive data to a single location suffers from several drawbacks in the coopetitive setting. First, transferring and maintaining a remote copy of large data incurs significant overhead especially if this data changes frequently. Second, this strategy may run afoul of regulations that forbid a database from being moved out of a certain perimeter. Third, the oblivious computation in Opaque crucially assumes that all communication happens within the same cloud; applying Opaque's algorithms to query execution across different parties connected by a wide-area network results in prohibitive overhead.

In this chapter, we propose Oblivious Coopetitive Queries (OCQ), a general framework for coopetitive analytics based on hardware enclaves, overviewed in Figure 2.1. Rather than requiring multiple parties' data to be aggregated to a single location, OCQ executes queries in a decentralized manner. OCQ develops efficient *oblivious query algorithms* (e.g., oblivious federated join) for the federated setting and a *schema-aware padding* mechanism, which combined prevent data leakage through (1) memory accesses to data inside the enclaves, and (2) network traffic patterns outside the enclaves, both within a local data center and in the wide area. OCQ also contributes an *oblivious planner*, which determines where to execute each operation and how to execute it, to minimize the

overhead while maintaining the oblivious security guarantees.

We implemented OCQ as an extension to Apache Spark SQL's Catalyst query planner and execution engine. We evaluate OCQ using SGX-enabled, geographically distributed clusters on a variety of synthetic benchmarks and find that OCQ is up to 9.9× faster than outsourcing all data and computation to a third party running Opaque. Compared to AgMPC [136], a state-of-the-art cryptographic framework for secure multi-party computation, OCQ is up to 219× faster.

OCQ's design addresses the following challenges:

Challenge 1: Oblivious queries in the wide area. The first step of distributed operators such as aggregations and joins is typically to shuffle the entire relation to colocate the appropriate records. For example, Opaque implements aggregation using an initial distributed sort based on the grouping attributes to colocate records that belong to the same group. However, in the coopetitive setting, this incurs record movement across the wide area, requiring the use of expensive security protocols to avoid leaking information.

Approach: Federated and oblivious planner. Our federated planner chooses operators that maximize the computation run at originating parties and ensures that communication across the wide area does not leak information about the input data. For example, while a high-cardinality aggregation that results in many groups would normally be implemented using an initial distributed sort, OCQ's planner instead chooses to compute partial aggregates at each party and shuffle those partial aggregates across the wide area, with padding to hide the number of groups from each party. The worst-case upper bound on the number of groups is often much smaller than the number of rows in the database, for example, for fields containing gender or age. This approach avoids exchanging un-aggregated records between parties. Key to performance is that parts of the computation running within a party's cluster that only touch that party's data need not be oblivious because the data is known to the party. Such local computation will still run inside the enclaves for integrity and authentication.

Challenge 2: Combining data of mixed sensitivities. Queries may consist of operators that combine slices of multiple parties' data. Because OCQ executes these operators at the parties themselves, and because parties may provide table-level sensitivity annotations, many relational operators in OCQ combine data of varying sensitivity levels. For example, one party may execute a join operator between its own data and a slice of sensitive data from other parties. Executing such operators using fully-oblivious algorithms would incur unnecessary overhead.

Approach: Mixed-sensitivity algorithms. OCQ introduces mixed-sensitivity operators, such as a mixed-sensitivity oblivious join algorithm based on the merge phase of bitonic sort that provides up to 2.5x speedup compared to a fully-oblivious join.

Challenge 3: Query planning with sensitive cardinalities. Query planners traditionally rely on statistics to choose among multiple plans. However, such statistics are sensitive in a coopetitive setting, as they reveal information about the distribution of each party's data. The plan chosen can leak information about the input statistics. Additionally, the cardinalities of intermediate relations may leak information, such as the selectivity of a filter. Cardinality leakage poses a further threat in the coopetitive setting than in the outsourced computation setting because a malicious party can

manipulate its input to extract information through cardinalities.

Approach: Schema-aware padding. As in previous work on Opaque [145], we take the approach of padding input and intermediate relations to publicly-known bounds to hide sensitive cardinalities. Our contribution is a scheme to refine these bounds by exploiting the likely presence of foreign key relationships between public and private relations in each party's schema to find tighter padding bounds for each operator. For example, a query to find all distinct disease diagnoses across multiple hospitals would typically involve padding to the number of patient diagnoses, while a foreign key relationship to a set of diseases would enable OCQ to pad to the possibly much smaller number of registered diseases. We introduce rules to propagate these bounds through the query plan. Conveniently, padding rules also obviate the problem of leakage via the choice of plan, because the resulting padding bounds provide exact cardinality information without leaking sensitive statistics.

2.2 Background

OCQ is designed to enable coopetitive analytics using hardware enclaves. Here we describe the coopetitive setting and provide background on the building blocks for OCQ.

2.2.1 Hardware enclaves

Hardware enclaves or trusted execution environments (TEEs) such as Intel SGX [94], AMD Memory Encryption [78], Keystone [82], Sanctum [39], MI6 [22] and others [133, 10] enable code to run in an isolated environment where other processes on the same host, including the OS and hypervisor, cannot tamper with its execution or access its memory. Enclaves also provide remote attestation, which allows the enclave to prove to a client that it is running the desired code and to establish a secure channel to a client. We discuss in Section 2.4 the enclave threat model we build OCQ on.

2.2.2 Oblivious algorithms

As we discuss in Section 2.4, enclaves suffer from side channels exploiting memory access patterns to data and traffic patterns. OCQ protects against these side channels with oblivious computation and appropriate padding. Oblivious algorithms aim to process data while ensuring that their memory accesses are independent of the contents of the data; this also implies that the network traffic patterns are also independent of data content. For example, basic matrix multiplication is oblivious, because its access pattern depends only on the size of the inputs and not their numerical values. In contrast, quicksort is not oblivious because its access pattern depends on the ordering of the data: in each iteration, records smaller than the pivot are swapped to one memory region, while records larger than the pivot are swapped to another. The choice of where to swap each record depends on the record contents.

Because sorting is at the heart of most database operators, efficient oblivious sorting algorithms are of particular interest. Single-machine oblivious sorting can be done using sorting networks that perform a fixed sequence of compare-exchange operations. Asymptotically more compare-exchange

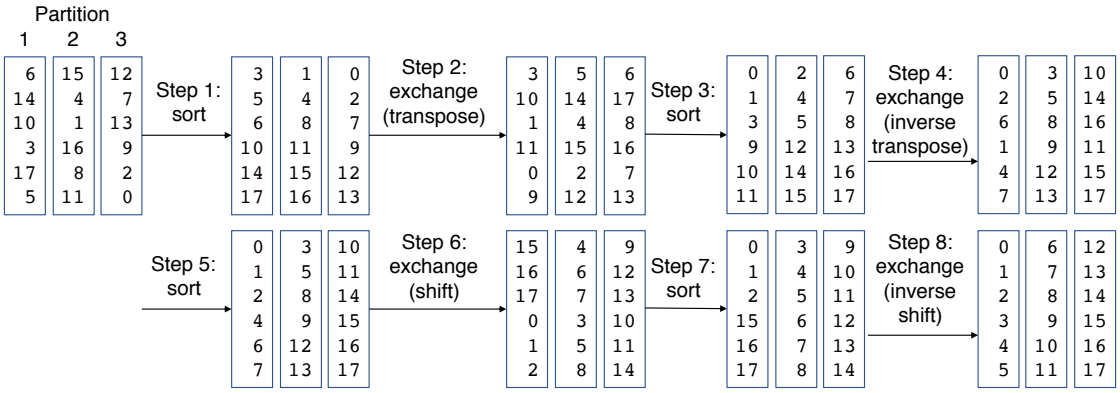


Figure 2.2: Illustration of column sort algorithm. Each column represents an encrypted partition. Column sort enables oblivious distributed sorting using four intra-partition sorts (steps 1, 3, 5, 7) and four data exchanges (2, 4, 6, 8) in fixed patterns. An attacker only sees exchanges of encrypted records. Since the pattern of exchanges is fixed, it cannot leak data contents.

operations are needed for oblivious sorting than for traditional sorting. An oblivious compare-exchange can be implemented via a comparison followed by a conditional swap of two equal-length buffers depending on the result of the comparison.

For data partitioned across multiple machines, oblivious sorting can be accomplished using a two-level sorting algorithm in which each partition is individually sorted using a sorting network, and records are sorted across partitions using an algorithm called column sort [85]. Column sort consists of a fixed sequence of data exchange and intra-machine sorting that uses only 4 shuffles, compared to $O(n \log^2 n)$ shuffles for a sorting-network-based distributed sort. It is thus well suited to oblivious distributed sorting, where it was previously applied by Opaque [145]. OCQ also uses it for sorting sensitive data; the algorithm is illustrated in Figure 2.2.

2.2.3 Spark SQL and Opaque

OCQ's planner and federated execution engine are built on Spark SQL [142, 11], a distributed SQL analytics framework, and Opaque [145], an extension of Spark SQL for secure outsourced computation via hardware enclaves.

Spark SQL offers distributed plaintext query execution. Queries can be written in SQL or an embedded Scala DSL called DataFrames. The user submits queries to the Spark SQL driver, which parses them into logical plan format. Spark SQL's extensible rule-based query planner, Catalyst, which also runs at the driver, optimizes these plans and generates a physical plan for execution on a Spark cluster. Catalyst is primarily rule-based, but offers limited statistics collection and cost-based optimization. The physical plan breaks the query into stages consisting of parallel tasks that are executed on workers. Each worker writes results to distributed storage or returns them to the driver.

Opaque extends Spark SQL to the untrusted cloud setting, where the driver is trusted but the workers are not. By extending Catalyst with encrypted operators which result in tasks that run inside

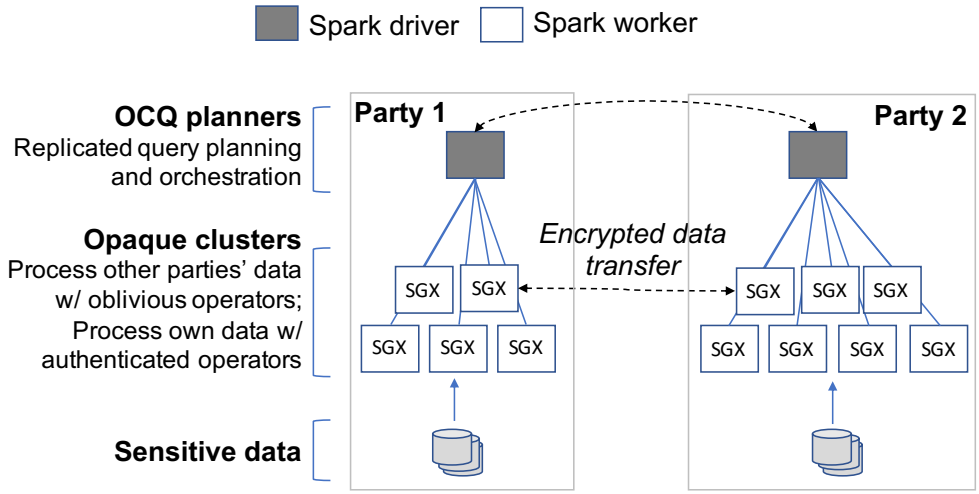


Figure 2.3: Architecture of OCQ. OCQ's replicated federated planner executes operators on Opaque and Spark clusters at each party. Sensitive data never leaves its originating party in plaintext.

SGX enclaves rather than in plaintext, it enables distributed queries on encrypted data.

OCQ's federated planner is an extension of Catalyst, and OCQ leverages Opaque to process encrypted data within a single party. This enables OCQ to inherit Spark SQL's query languages and optimizations, and Opaque's secure distributed query processing. However, OCQ must implement rules and orchestration logic specific to secure *federated* queries.

2.3 Architecture

Figure 2.3 shows OCQ's architecture. Each party maintains a Spark cluster with at least one hardware enclave-enabled machine, on which Opaque tasks are scheduled. OCQ's query planner is deterministic and runs outside the enclave at every party. This is because our query planner builds on Spark SQL's planner, which is a large Scala codebase that would significantly broaden the enclave's attack surface and require heavyweight sandboxing techniques. To reduce coordination between parties at query time, each party runs a replica of OCQ's federated planner; we describe the mechanism for verifying the plan's integrity in Section 2.3.1. Each planner replica maintains an audit log of all queries issued by any party.

The role of each party's Opaque clusters is to (1) give assurance that the computation at each party happens correctly, and (2) safely mix multiple parties' data. Therefore it is essential for enclave code to be trusted by all parties. OCQ accomplishes this by granting all parties the ability to invoke pre-approved routines on all enclaves, but ensuring that each enclave verifies that the deployed code is approved by all the parties via remote attestation.

In OCQ, parties own different tables; a logical table consisting of rows from different parties can be implemented using union. Parties annotate their tables as either public or sensitive. The query planner determines the sensitivity level of intermediate results using sensitivity propagation rules

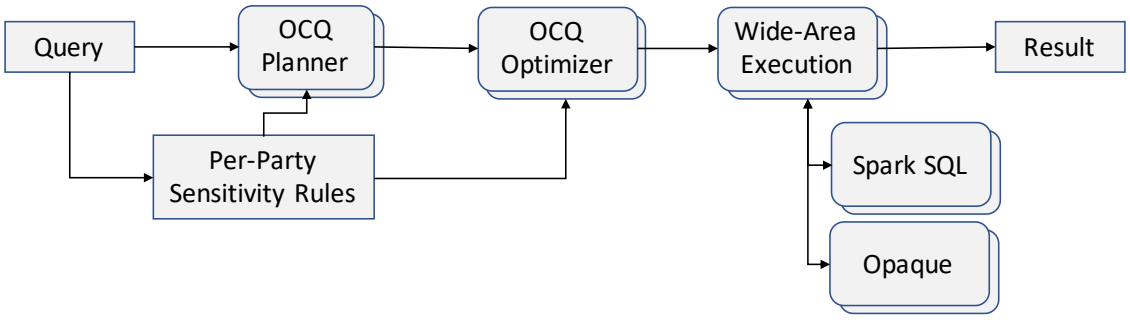


Figure 2.4: Lifecycle of a query. Queries and sensitivity annotations are submitted to OCQ's federated planner and optimizer. The resulting federated plan contains operators running at different locations across the federation, and satisfies all parties' sensitivity annotations. The plan executes securely, and the user only learns the result and cannot access sensitive intermediate relations. Stacked blocks indicate that a component is present at each party.

discussed in Section 2.5.

2.3.1 Setup phase

Parties must share the cryptographic hash of each encrypted partition of each of their input relations with all other parties. Once the setup phase has completed, no party can change its input data, and the enclaves will ensure this.

Parties setup their enclaves using OCQ's code, and perform remote attestation amongst all these clusters. Consider a logical enclave at each party (which can be implemented via a cluster of enclaves). The result of this stage is:

- Every party and its enclave know the public keys of every party, the schema and sensitivity of every table of each party, and the hashes of each party's encrypted data.
- Each enclave checks that the enclaves at the other parties were correctly setup with OCQ with the same information.
- The enclaves agree on symmetric keys for a secure channel amongst themselves.

Queriers may attempt to submit malicious queries designed to extract sensitive data, and a compromised planner replica may produce a physical plan that reveals sensitive information. We prevent both of these by requiring all parties to agree on the allowed set of queries, the resulting query plans, and size bounds for sensitive input and intermediate data sets. The agreed-upon configuration is represented as a set of DDL statements, queries, and physical plans that is signed by all parties and passed to each party's instance of the OCQ federated planner and enclave. Before signing the configuration, each party should check that it matches expectations.

2.3.2 Query lifecycle

Figure 2.4 shows the lifecycle of a query. A user submits a query to a federated query planner replica, which broadcasts it to all the others, one planner instance per party. Each planner first checks that the query conforms to the set of approved queries and then performs the same planning and optimization steps, deterministically generating a federated plan with operators running at different locations. The generated plan satisfies all parties' sensitivity annotations and performs as much computation as possible in plaintext at each party. The party signs the plan, and enclaves will execute only plans signed by all the parties. Each planner runs the relevant operators at its local Spark and Opaque clusters.

Special data movement operators trigger data exchange with other parties across the wide area, or between one party's Spark and Opaque clusters, when allowed by the sensitivity annotations. The final operator collects the results back to the originating replica and returns them to the user.

A party can observe network traffic generated by computation on other parties' data when that computation is run within their Opaque cluster, such as during final aggregation. To prevent size leakage in this case, OCQ automatically determines appropriate public upper bounds for all intermediate results and ensures that each operator pads its output to the appropriate bounds. Section 2.5.4 describes this process in more detail.

Our integrity mechanism for ensuring that a malicious party cannot tamper with the integrity of the computation or input data is similar to that of Opaque [145]'s self-verifying computation, so we only describe it at a high level. At query time, the signed physical plans are loaded into the enclaves, which check that all parties signed every plan. When an enclave is requested to execute part of a query plan, it verifies that each of the inputs to the plan fragment were either authentic input data or were generated by the expected child plan, depending on the expected input source. To check that the input is authentic, when scanning a party's input data, the scanning enclave checks the partition's hash against the party's hash from setup to ensure that the party has not tampered with the input data. If the hashes do not match, the enclave signals failure and the query will abort. After executing the plan fragment, the enclave certifies that its output was correctly produced by running the requested plan fragment.

2.4 Threat model and security guarantees

2.4.1 Abstract enclave model

OCQ considers an attacker who can see memory accesses to data and/or messages sent over the network. Such an attacker arises from a large class of attacks: attacks leveraging page faults [139, 27, 135], the branch predictor [84], cache timing [123, 24], the memory bus [81], network traffic patterns [106], and others. Oblivious algorithms are impressively effective against such a large variety of attacks [68] because they address the core leakage of these channels: memory accesses based on sensitive data. OCQ contributes oblivious algorithms for the federated analytics setting to thrwart such attacks. We group the attackers in two categories:

- A **network attacker** that sees all network traffic but has no access to the machines. In particular, this attacker does not see any memory accesses inside any of the machines.
- A **malicious party attacker** that sees all memory accesses made by enclaves in addition to the network traffic. For this party we assume the oblivious row-level conditional-exchange primitive from Section 2.2.2.

OCQ builds on top of an *abstract* enclave model; OCQ's design is not tied to Intel SGX even if our prototype implementation in Section 2.7 builds on it. There are many different proposals for hardware enclaves, such as Intel SGX [94], AMD Memory Encryption [78], Keystone [82], Sanctum [39], MI6 [22] and others [133, 10], with more upcoming as researchers make rapid progress towards more secure hardware enclaves. OCQ assumes that the attacker cannot access or undetectably modify data or code inside the hardware enclave, that it cannot exploit side-channels different from the memory addresses discussed above, subvert the remote attestation process or otherwise the integrity or secrecy of the enclave; if the attacker could, OCQ does not protect against such attacks. Such side-channel attacks indeed exist in some enclave implementations (such as in Intel SGX); addressing them should be done at a different level of abstraction than OCQ operates at, for example, through better enclave design. For instance, Intel SGX suffers from various vulnerabilities or side channels, such as attacks based on speculative execution [26, 33, 122], power consumption [102, 129], rollback [113], intra cache line memory accesses [140, 99], denial-of-service attacks [74, 62], and others (e.g., [138, 83]). Many defenses or mitigations have been proposed, such as for closing hyperthreading-based attacks [108, 34], rollback defenses [23, 92], and importantly, improved enclave designs that remove a wide array of the attacks above, such as KeyStone [82] or MI6 [22] (the latter even protects against speculative-execution attacks). We hope that OCQ's design will be ported to better and better enclaves as research progresses on this front.

We remark that our implementation prototype of OCQ described in Section 2.7 focuses on obliviousness with respect to *data* at the *cache-line* granularity. While OCQ's oblivious algorithms are oblivious even when it comes to accesses to OCQ's pseudocode, we have not ensured that our implementation and the generated binary preserve this property. Also, while OCQ's oblivious algorithms could be applied at the intra-cache-line granularity [140, 99], our prototype implementation does not implement them at this level. Both of these can be addressed in the implementation with existing mechanisms (e.g., [108, 34, 88, 61, 40]) at a performance cost.

2.4.2 Party threat model

In the coopetitive setting, a malicious party could attempt to tamper with the other parties' data during joint computation, or it could attempt to inspect other party's data using the attacker capabilities we discussed in §2.4.1. A party can also observe and modify network traffic generated by such outsourced computation. Given the assumption of an abstract enclave model, OCQ guarantees integrity of outsourced operators and data.

Each party is free to input data of its choice, and OCQ does not protect against low-quality or maliciously-crafted data. OCQ ensures the integrity of each party's computation and data after the

data has been inputted. The parties, if they wish to, could include checks for each other's data in the queries given to OCQ.

Query results often leak some information about the data from which they were computed. This is why we require the parties to agree on what queries they permit to run on their data and whose results they are agreeing to release. OCQ ensures that all parties agreed to running some query before running that query and releasing the result. Deciding which queries are safe to run is outside the scope of our system, and is largely an unsolved research problem. Nevertheless, existing work in differentially private analytics [76, 77, 16] (discussed further in §2.9.4) and inference detection [66, 45, 116] could aid parties to transform the queries into a safer form or to detect particularly revealing queries.

2.4.3 Security guarantees

OCQ guarantees obliviousness for sensitive tables, namely that there is no information leakage about the sensitive data content from the trace of memory accesses and traffic messages other than size, schema, and query information. We use a standard definition of obliviousness, which states that the *trace* of memory accesses and messages can be simulated without access to the data, while only knowing a bound on the data size, the schema, parties' configuration, and the query plans to execute.

The proof that OCQ meets this guarantee follows from the observation that for all our protocols (query planner, overall query plan, and individual operators), all the accesses to memory and the schedule of messages on the network are performed according to a *predefined schedule*, fixed ahead of time before the data is input, and which is determined according to our planner's rules based on data size, schema, query and party's configuration.

Like in much prior work in oblivious computation, our formalism captures only addresses and lengths, and not the actual data content. The reason is that hardware enclaves as used in OCQ and other works, encrypt the content and re-encrypt it upon every access.

We remark that OCQ's query planner runs entirely using non-sensitive information to produce a physical plan; hence, the planning process is oblivious by definition.

We now focus on showing that the execution of each physical plan of a query is oblivious w.r.t. sensitive tables. OCQ's definition of obliviousness differs from that of Opaque because there are multiple parties in addition to the network attacker. A malicious party attacker may see the content of its own sensitive tables and perform non-oblivious computation on them, but not the content of other parties' sensitive tables. We formalize this by constructing two types of simulators:

- $\text{Sim}_{\text{party}}$ has access to a party's sensitive data, but does not have access to the sensitive data of other parties, yet must simulate this party's memory and network access patterns.
- Sim_{net} does not have access to any party's sensitive data and must simulate the network communication patterns among all parties.

The fact that the simulators can simulate the memory and network patterns without seeing all the sensitive tables implies that these patterns do not leak information about these sensitive tables.

Let D be the set of tables of all parties. As we discussed in Section 2.3, tables are of two types: sensitive and public. Let S_i be the set of tables of party i that are sensitive (including those OCQ marks as sensitive after propagating sensitivity as discussed in Section 2.5). Let Public(D) be all the non-sensitive information about the datasets, such as the content of all public tables and publicly known metadata about the sensitive tables, such as the names and owners of the tables, the names and schema of columns, an upper bound on the number of rows in each table, the set of unique key and foreign key constraints, and others. This metadata does not include the actual values in any column.

Let P be the set of publicly known cluster configurations of the parties, including the number of workers and their IP addresses, the untrusted memory size and EPC size of each machine, the oblivious sorting block size, etc. Let q be a query with any user-specified padding bounds (Section 2.5.5). Let $\text{Trace}_i(D, P, q)$ be the access pattern trace visible to the malicious attacker at party i: an ordered sequence of memory accesses of the form `(read/write, addr, length)` and network messages of the form `(dst, length)`. These do not contain timestamps because OCQ does not protect against timing attacks. $\text{Trace}_{\text{net}}(D, P, q)$ is the ordered sequence of network messages across all parties.

Theorem 1. *For all parties i, datasets D, with party i's sensitive tables S_i, for all cluster configurations P, for all queries q there exist polynomial-time simulators* $\text{Sim}_{\text{party}}$ *and* Sim_{net} *such that*

$$\forall i, \text{Sim}_{\text{party}}(\text{Public}(D), i, S_i, P, q) = \text{Trace}_i(D, P, q),$$
$$\text{Sim}_{\text{net}}(\text{Public}(D), P, q) = \text{Trace}_{\text{net}}(D, P, q).$$

We provide a proof sketch in Section 2.6.

2.5 Query Planning

We next explore OCQ's federated query planner, which finds query plans that satisfy security properties while minimizing the scope of expensive oblivious operators.

2.5.1 Overview

The federated query planner accepts sensitivity annotations on each party's base tables. It supports two sensitivity levels:

- *Public*: the table can be processed at any site with integrity verification. Confidentiality is not required, and data contents and cardinality may safely be exposed.
- *Sensitive*: if exported from the originating site, the table must be processed with confidentiality and integrity verification, including protection against access pattern side channels and cardinality leakage.

When executing a query, the planner propagates sensitivities through the plan to determine the sensitivity level of each intermediate result, such as a set of partial aggregates. OCQ uses the same two-part sensitivity propagation scheme as Opaque. First, since tables could be correlated via their relationships (e.g., foreign keys), OCQ uses second-path analysis [66] on the user-provided sensitivity annotations to capture these correlations: user-specified base tables are initialized as sensitive, and all tables reachable from a sensitive table via primary–foreign key relationships are recursively marked as sensitive as well. Second, OCQ assigns each operator a sensitivity level determined by the sensitivity levels of its inputs. Operators that process more than one input relation (e.g., join) receive the highest sensitivity level of all their inputs.

The federated query planner uses these two sensitivity levels to determine where to execute each operator. Recall that each party has an Opaque site/cluster. Operators may execute in two locations: federated or single-site. The federated operators execute in every site in parallel, whereas the single-site ones execute only at one party. For example, for an aggregation, each site can perform a partial aggregation within their site using the federated operator and the results are then sent to the querying party for final aggregation, which occurs using the single-site operator. OCQ supports the following operator execution modes:

- *Federated.* The operator executes partitioned and encrypted/authenticated in each site's Opaque cluster using SGX. Datasets are encrypted and authenticated for integrity verification, but operators will reveal data cardinality and may leak data contents through side channels.

- *Federated-Oblivious.* The operator executes partitioned and encrypted in each site's Opaque cluster using oblivious algorithms in SGX to hide access pattern side channels. The operator reveals nothing beyond the cardinality of the input, such as filter and join selectivities.

- *Single-Site-Oblivious.* The operator executes obliviously with Opaque at the site where the query originated. This provides the same security guarantees as Federated-Oblivious and is used for final aggregation on sensitive data.

- *Single-Site.* The operator executes encrypted using Opaque at the site where the query originated. This provides the same security guarantees as the Federated mode and is used for final aggregation on public data.

The query planner ensures data of a particular sensitivity level is never processed using an operator with insufficient protections. For example, Sensitive data may be processed with a Federated operator if it has not left its originating site, but after an exchange, the same data must be processed using Federated-Oblivious or Single-Site-Oblivious operators only. The planner can always produce a secure plan for any supported query by running all operators in the Single-Site-Oblivious mode with naive worst-case padding, but the resulting overhead can be prohibitive so its goal is to find secure plans with much lower overhead when possible.

In the remainder of this section, we describe our query planning algorithm and rules (Section 2.5.2, Section 2.5.3), then discuss how the planner hides intermediate cardinalities (Section 2.5.4). The strict sensitivity levels and the use of padding together simplify physical operator selection. This

traditionally depends on accurate cardinality estimation, which is much easier in OCQ than in traditional databases due to OCQ's use of padding. OCQ introduces padding rules that exploit foreign key constraints in the database schema to minimize padding overhead.

2.5.2 Algorithm

Respecting the constraints described above, the federated query planner applies rules to produce a physical plan that specifies the necessary algorithms and data movement for all three levels of the federation: between parties, within each party, and within each machine. Planning occurs as follows:

1. Obtain the logical plan for the query using Spark SQL.
2. Apply rules to transform the logical plan into a federated physical plan respecting sensitivity annotations and with "ShipTo" operators indicating data movement.
3. Eliminate redundant ShipTo operators.
4. Insert padding operators based on the rules in Section 2.5.4.
5. Execute the federated plan and return the results.

The rules used in step 2 apply recursively to the logical plan in bottom-up order starting with the base tables. For each logical operator (e.g., Project, Filter, Join, Aggregate) and its input plans, the rules produce a physical sub-plan that respects the operator's sensitivity level and avoids unnecessary encryption or data movement overhead.

OCQ's rules require operators to be planned in order of execution, because it uses the execution mode chosen for earlier operators in determining the execution mode for later operators. This is implemented using a postorder traversal on subplans referencing sensitive data.

2.5.3 Query planner rules

We now examine rules for the four common logical operators (Project, Filter, Join, and Aggregate). We express rules in Scala's pattern match syntax, where each rule is a `case` that may match the logical operator being planned. Rules can specify subtyping constraints with `:` syntax. We use these constraints to discriminate based on the execution mode (abbreviated as Fed, FedObl, SSObl, and SS) of an operator's inputs. Each rule ensures the resulting physical operator respects sensitivity levels, so the entire plan also will.

To illustrate how OCQ converts a query into a physical plan, take the example of Query 1 in Table 2.1. For each logical operator that matches a rule, Scala binds the operator's contents to parameters listed after the `case` keyword. It then executes the body of the case, which comes after the => arrow, and the planner uses the return value as the physical plan for the given logical operator. For example, the logical operator `Filter(diagnosis, diag="c. diff")` in Query 1 (Table 2.1), will be mapped to OCQ's physical operator `FedEncFilter[diag="c. diff"]`. We do not describe what each one of OCQ's physical operators run (because they are many), but instead describe only the more novel operator algorithms in §2.6.

Project. The input to the projection is referred to as `child`, and the projected columns as `p`. Because a projection never requires data movement and does not leak access patterns, the resulting operator uses the same execution mode as its input.

```
case Project(p, child: Fed) => FedEncProject(p, child)
case Project(p, child: FedObl) => FedOblProject(p, child)
case Project(p, child: SSObl) => OblProject(p, child)
case Project(p, child: SS) => EncProject(p, child)
```

Filter. The input to the filter is referred to as `child`, and the filter predicate as `f`. As with projection, filtering never requires data movement. However, unlike projection, filtering leaks access pattern information by default, so depending on the execution mode and sensitivity level of the input, we use an oblivious filter operator to hide which input records matched the predicate.

```
case Filter(f, child: Fed) => FedEncFilter(f, child)
case Filter(f, child: FedObl) => FedOblFilter(p, child)
case Filter(f, child: SSObl) => OblFilter(p, child)
case Filter(f, child: SS) => EncFilter(p, child)
```

Join. OCQ currently only supports inner joins. The inputs to the join are referred to as `left` and `right`, and the join columns as `c`. Joining does require data movement, and we choose between a broadcast join where one input is broadcast to all parties and joined separately with each portion of the other input, and a single-site join where both inputs are brought to the querier's cluster. Both types of joins can be executed with or without obliviousness; we additionally implement a mixed-sensitivity broadcast join (Section 2.6.1). For brevity, we only list half the rules for Join; the other half are the same up to swapping left and right. When multiple join rules apply, such as when there is a choice between broadcasting the left side and the right side, the planner currently chooses one of them arbitrarily, but a cost-based planner could consider both and choose the lower-cost option. In addition, we benefit from Spark SQL's existing planner rules. For example, Spark SQL's broadcast exchange reuse ensures that when a plan indicates that the same relation should be broadcast more than once, it will only be shipped over the WAN once.

```
case Join(c, left: Fed, right: Public) =>
  FedEncJoin(c, left, BcastToFed(right))
case Join(c, left: Fed, right: Sensitive) =>
  FedMixedSensJoin(c, left, BcastToFed(right))
case Join(c, left: FedObl, right: Public) =>
  FedMixedSensJoin(c, left, BcastToFed(right))
case Join(c, left: FedObl, right: Sensitive) =>
  OblJoin(c, EncCollect(left), EncCollect(right))
case Join(c, left: SSObl, right: Public) =>
  MixedSensJoin(c, EncCollect(left), EncCollect(right))
case Join(c, left: SSObl, right: Sensitive) =>
  OblJoin(c, left, EncCollect(right))
```

Aggregate. The input to the aggregation is referred to as `child` (which is a query subplan), the grouping attributes as `g`, and the aggregation attributes as `a`. Partial and final aggregates are assigned the same sensitivity level as the input data. The attributes resulting from partial aggregation are referred to using `partial(a)` and the attributes from final aggregation as `final(a)`. The execution mode of partial aggregation is the same as its input, while final aggregation always occurs at the querier's site in SSObl or SS modes. The `with` keyword indicates the child's execution mode as well as its sensitivity.

```
case Aggregate(g, a, child: Fed with Public) =>
  EncAgg(g, final(a),
    EncCollect(FedAgg(g, partial(a), child)))
```

This rule applies to an aggregation over `Public` data which will reside in a `Fed` manner as a result of running the child subplan. Because the data is public, we can execute both the partial and the final aggregation in `Enc` mode (without oblivious operators).

```
case Aggregate(g, a, child: Fed with Sensitive) =>
  OblAgg(g, final(a),
    EncCollect(FedAgg(g, partial(a), child)))
case Aggregate(g, a, child: FedObl) =>
  OblAgg(g, final(a),
    EncCollect(FedOblAgg(g, partial(a), child)))
```

Both rules apply to sensitive federated data. In the first case, each sensitive data slice has not left its originating party, while in the second case, the sensitive data has been commingled with other parties' sensitive data and is protected by obliviousness in addition to encryption. Therefore, in the first case only the final aggregation needs to be oblivious, while in the second case, both the partial and final aggregation must be oblivious. The remaining rules are as follows:

```
case Aggregate(g, a, child: SSObl) => OblAgg(g, a, child)
case Aggregate(g, a, child: SS) => EncAgg(g, a, child)
```

Query planning example. Table 2.1 shows the results of OCQ's federated planning on two sample medical queries [15]. The queries refer to two tables: a `diagnosis` table containing patient SSNs and the diseases they were diagnosed with, and a `medication` table containing patient SSNs and the medications they were prescribed. Given these two relations, Query 1 computes comorbidity of the disease *c. diff*: the most common diseases that *c. diff* patients are also diagnosed with. Query 2 counts the number of patients with heart disease who were prescribed aspirin. Tables and intermediate results containing identifiable patient information (here SSN) were specified as Sensitive; tables without SSNs but with more than one column as Sensitive to prevent correlation attacks; and tables with only one non-identifiable column as Public.

For Query 1, the planner runs the initial Filter operator in Federated mode. Subsequently, only the `diag2` column of the result is needed for the Aggregate operator, so Spark SQL automatically inserts a projection to drop the other columns. The resulting table is collected to a single site for the final aggregation and sort.

Query 1 (comorbidity of *c. diff*):

```
Sort(
  Aggregate(
    Join(
      Filter(diagnosis, diag="c. diff"),
      Project(diagnosis, diag as diag2),
      col="patientSSN"),
    groupby="diag2", agg="count"), col="count")
```

Federated plan for Query 1:

```
OblSort [diag_count]
  OblAgg groupby[diag2] agg[count(*) as diag_count]
    OblCollect
      FedOblProject [diag2]
        FedMixedSensJoin [patientSSN]
          BcastToFed
            FedEncFilter [diag="c. diff"]
              FedEncProject [diag]
                FedEncScan diagnosis
          FedEncProject [diag as diag2]
            FedEncScan diagnosis
```

Query 2 (aspirin count):

```
Count(
  Aggregate(
    Join(
      Filter(diagnosis, diag="heart disease"),
      Filter(medication, med="aspirin"),
      col="patientSSN"),
    groupby="patientSSN")
```

Federated plan for Query 2:

```
OblCount
  OblAgg groupby[patientSSN]
    OblCollect
      FedOblProject [patientSSN]
        FedMixedSensJoin [patientSSN]
          FedEncFilter [diag="heart disease"]
            FedEncScan diagnosis
          BcastToFed
            FedEncFilter [med="aspirin"]
              FedEncScan medication
```

Table 2.1: Example queries and resulting federated plans. Queries (top) are specified in Spark SQL's logical plan notation, which is similar to relational algebra. The resulting plans (bottom) are specified in Spark SQL's physical plan notation with OCQ's physical operator names. Nesting indicates a child relationship; sub-plans nested under a physical operator provide input to that operator. Physical operators take parameters, listed on the same line as the operator name. Most operator parameters are expression lists, listed in square brackets.

For Query 2, the planner likewise runs the initial filters in Federated mode. As in Query 1, the right side is Public, so the planner uses a mixed-sensitivity broadcast join. The oblivious aggregation returns the number of distinct patients.

2.5.4 Determining padding upper bounds

Sensitive operators must pad their output to avoid leaking information about the input. For example, a bank may want to hide how many customers it has, so the data sizes of any cross-site communication must be padded to a bound greater than the number of customers. Alternatively, to hide the number of customers with revenue >$1 million, computation downstream of all revenue-based filters must be padded.

The federated query planner ensures queries' intermediate cardinalities do not leak information about sensitive attributes. The final cardinality is handled the same as the intermediate cardinalities, depending on which parties can see the final result. After producing an unpadded plan using the rules above, it adds padding to Sensitive operators that could leak information about the table's contents. This section describes this rule-based process using the same rule syntax as Section 2.5.3. Though these rules match physical operators, we use logical plan names in some pattern specifications to denote matching all physical operators implementing a logical operator. Additionally, we refer to certain specific join types such as referential integrity inner equi-joins as a shorthand for pattern specifications that check that these constraints are satisfied. Finally, we use Scala's `@` operator to denote binding the physical operator under consideration to a named variable.

Base tables. Base tables are padded using tiered padding to hide their exact cardinality, encapsulated by the `round` function, specified by the parties. We refer to the table scan operator as `t`. The rule wraps `t` with a `Pad` operator that inserts dummy rows to inflate the result cardinality. The base table data must already be padded to an equal or greater bound when writing it to disk to avoid revealing the true cardinality through the file size. This enables this operator to make dummy accesses to the input when generating the dummy output rows.

```
case t @ TableScan() => Pad(round(t.cardinality), t)
```

Filters. Most filters must be padded to the input size to avoid leaking selectivity. The `else` clause of the rule below transforms the filter into a projection that adds a boolean field with the value of the filter predicate using Scala's `:+` operator, which appends this column to the existing columns. Later operators will use this tombstone-like field to determine whether or not to include the record in their operations.

When at most one record is being selected, such as to extract the record of a single patient based on SSN, padding to the input size would be very wasteful. OCQ identifies this case by checking if the predicate is an equality comparison against a unique key using the `uniquelyReferences` method. The `if` clause below pads the result to a cardinality of 1 to avoid leaking whether or not the predicate matched a row. Within this rule, which branch is taken depends only on the schema and query, not the underlying data, so it does not reveal any new information to an attacker.

```
case f @ Filter(pred, child) =>
  if (pred.uniquelyReferences(child.keys)) Pad(1, f)
  else Project(child.cols :+ pred, child)
```

Joins. Arbitrary joins must be padded to the product of table sizes, but common join types have much smaller upper bounds. Unique key equijoins are a very common join type where one side's join attribute is known unique. We pad these joins to the size of the other table; the unique key ensures each record in the latter matches at most one record in the former. For brevity, we omit the code listing for this rule.

Aggregations. Arbitrary aggregations must be padded to the input size, because each input record could belong to a different group. However, foreign key constraints let us refine this bound. For example, an aggregation over patients' diseases can yield at most one row per disease if there is a foreign key constraint between the (sensitive) patient diagnosis table and the (public) disease table, because the foreign key constraint implies that all patients' diseases correspond to a record in the disease table. Without the foreign key constraint, we must pad the aggregate output to the number of patient diagnoses, which could be large. With the constraint, we can instead pad to the number of diseases.

We search for such foreign key constraints using `publicTableKeys` in the rule below. If there is a matching foreign key, the `Some(...)` case determines the appropriate cardinality, using `min` to ensure the new bound is smaller. Otherwise, the `None` case pads to the input cardinality.

```
case a @ Aggregate(groupCols, aggExprs, child) =>
  publicTableKeys.find(groupCols.output) match {
    case Some(tbl, key) =>
      Pad(min(tbl.cardinality(key), child.cardinality), a)
    case None =>
      Pad(child.cardinality, a) }
```

2.5.5 Pre-specified padding bounds

Automatic padding bound search cannot always find the optimal bound. First, it assumes that while the exact cardinality of a base table is sensitive, rounded cardinalities are safe to share. Yet, for some tables, even an order-of-magnitude approximation of the cardinality is sensitive. Second, it relies on foreign key constraints to determine padding bounds for individual columns. However, many columns use implicit domains without foreign key constraints, such as ages, genders, letter grades, salaries, and ZIP codes. Third, most relational operators cannot reduce the padding bounds of their output, potentially resulting in large slowdowns.

We therefore support pre-specified padding bounds in schemas and queries. When defining the shared schema, parties can specify cardinality bounds for any table or column. When defining the allowed queries, a query may contain padding bounds for any intermediate result, for example the result of a filter expected to be highly selective. We exposed this functionality using extension

methods on Spark SQL's DataFrame API. Below is an example of specifying table and column cardinality bounds and a bound for a filter operator.

```
val diseaseDF = spark.load(".../disease/")
  .sizeBound(70000)
val employeeDF = spark.load(".../employee/")
  .colBounds("salary" -> 20, "addressZIP" -> 42000)
val singleEmployee =
  employeeDF.filter($"name" = "John Doe").sizeBound(1)
```

In case a query-specific operator underestimates the actual output cardinality, OCQ silently truncates the result to avoid leakage. If the parties wish to know when this has occurred, they can express this using subqueries.

2.6 Coopetitive algorithms

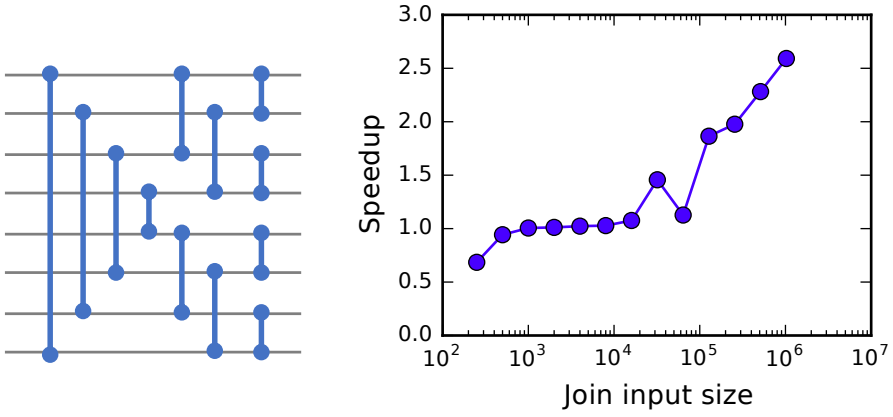
The coopetitive setting affects the performance of secure operators because it implies mixed-sensitivity computation and wide-area communication. Mixed-sensitivity computation occurs when a party combines its own data with sensitive data from another party. Wide-area communication occurs for operators such as aggregation and join that require combining data from multiple parties. We describe two algorithms that leverage the coopetitive setting to reduce the amount of oblivious computation needed compared to previous approaches.

2.6.1 Mixed-sensitivity join

Like Opaque's oblivious join algorithm, OCQ performs oblivious joins using bitonic sorting networks, which have traditionally been used in databases for SIMD parallelism [14] but which we use to protect against access pattern side channels. However, unlike in Opaque, oblivious joins in OCQ are very likely to be of mixed sensitivity. For example, a federated join between two Sensitive relations will be executed as an oblivious partial join at each party between that party's slice of one relation and the entire federation's records for the other relation. In this case, it is unnecessary to handle the former relation obliviously, since the owner of that relation is also the party processing it.

We therefore introduce a mixed-sensitivity join algorithm. We refer to the former relation as the non-sensitive relation and the latter as the sensitive relation. First, the non-sensitive relation is sorted using a conventional external sort. Next, the sensitive relation is obliviously sorted using the algorithm described in Section 2.2.2. The two sorted relations are then merged using an oblivious bitonic merge, illustrated in Figure 2.5a.

When joining two relations of equal size, asymptotic analysis shows that mixed-sensitivity join represents a constant-factor improvement. If the two relations contain $\frac{n}{2}$ elements each, an oblivious sort of the union requires $O(n \log^2 n)$ comparisons, while the mixed-sensitivity algorithm requires $O(\frac{n}{2} \log \frac{n}{2})$ comparisons for the conventional external sort, $O(\frac{n}{2} \log^2 \frac{n}{2})$ comparisons for the sensitive-relation oblivious sort, and $O(\frac{n}{2} \log n)$ comparisons for the bitonic merge. Figure 2.5b demonstrates an empirical speedup due to mixed-sensitivity join of up to 2.5× for large inputs. In



(a) Bitonic merge (b) Speedup from mixed-sensitivity join

Figure 2.5: (a) Bitonic merge network for 8 elements. (b) Speedup from mixed-sensitivity join compared to standard oblivious join for two relations of equal size. The x axis indicates the size of each relation in number of records. For small joins, other costs dominate, but once each relation contains more than 10^5 records, the mixed-sensitivity join shows a speedup of up to 2.5× compared to conventional oblivious join.

addition, when the sensitive relation is relatively small, mixed-sensitivity join becomes arbitrarily faster than a standard oblivious join.

2.6.2 Coopetitive aggregation

Opaque implements aggregation using an initial distributed oblivious sort based on the grouping attributes to colocate records that belong to the same group. However, in the coopetitive setting, this results in excessive wide-area data movement. Instead, we implement aggregation by computing partial aggregates at each party and sorting only those partial aggregates across the wide area, with padding to hide the exact number of groups from each party. The partial aggregates from each party are padded as described in Sections 2.5.4 and 2.5.5. The final aggregation is then performed as in Opaque, with a boundary processing step, a parallel scan over the sorted partial aggregates to produce final aggregates and dummy records, and, if a user-specified padding bound on the output is provided, an oblivious sort and filter to remove the appropriate number of dummies.

The speedup from this approach comes from avoiding the initial global oblivious sort in favor of an oblivious sort over the partial aggregates. When the bound of the number of groups is small, this produces a substantial performance gain.

2.6.3 Obliviousness proofs

Now that we presented OCQ's algorithms, we proceed to sketch the proof of its obliviousness, formulated in Section 2.4.3.

Proof of Theorem 1. In this proof, we will invoke the simulators for the oblivious building blocks that previously existed: bitonic sort, bitonic merge, column sort primitives, and the Opaque operators in oblivious-pad mode. A query q can consist of different tasks. It suffices to prove that the simulators can simulate the trace for each task. Here, we present simulators for the more complex algorithms OCQ contributes: mixed-sensitivity join and and coopetitive aggregation. For each physical operator O in the physical plan, the planner rules ensure that O runs in oblivious mode if the inputs to O contain any sensitive table owned by a party other than i.

Without loss of generality, consider party i. Recall that mixed-sensitivity join occurs between party i's own input A_i and a slice of the other parties' input B_i. If O is a mixed-sensitivity join, it proceeds as follows:

1. $\text{Sim}_{\text{party}}$ sorts A_i using quicksort because it receives A_i as input. Let $\text{Public}(A_{i,\text{sorted}})$ be the public metadata of the sorted result.

2. $\text{Sim}_{\text{party}}$ invokes the simulators for bitonic sort and column sort on $\text{Public}(B)$ to simulate an oblivious distributed sort of B on the equijoin keys. Let $\text{Public}(B_{i,\text{sorted}})$ be the metadata of the sorted result.

3. $\text{Sim}_{\text{party}}$ invokes the simulator for bitonic merge on $\text{Public}(A_{i,\text{sorted}})$ and $\text{Public}(B_{i,\text{sorted}})$. Let $\text{Public}(U_{i,\text{sorted}})$ be the metadata of the sorted union.

4. $\text{Sim}_{\text{party}}$ invokes Opaque's simulator for oblivious padded join on $\text{Public}(U_{i,\text{sorted}})$ with the padding bound specified in the query.

Sim_{net} runs similarly to $\text{Sim}_{\text{party}}$: it performs the last three steps above.

For coopetitive aggregation, if O is the partial phase of coopetitive aggregation at party i, let the padding bound for O be b. $\text{Sim}_{\text{party}}$ for party i extracts the data of the sensitive input A_i and executes a conventional hash aggregation on it, then pads its size to b. Let $\text{Public}(\text{Agg}(A_i))$ be the metadata of the padded partial aggregates.

If O is the final phase of coopetitive aggregation, $\text{Sim}_{\text{party}}$ for party i and Sim_{net} invokes the simulator for the bitonic sort and column sort on $\bigcup_{\forall i} \text{Public}(\text{Agg}(A_i))$. Sim_{net} then invokes Opaque's simulator for the oblivious padded aggregation on this metadata.

□

2.7 Implementation

We implemented OCQ on top of Intel SGX and as an extension to Apache Spark SQL's Catalyst query planner and execution engine using 2,000 lines of Scala code. OCQ builds on a version of Opaque we modified, which uses 11,000 lines of C++ enclave code and 3,000 lines of Scala code. No code changes to Spark SQL were required; both OCQ and Opaque extend Catalyst only by adding rules and strategies. The schema-aware padding requires that tables be annotated with primary and foreign key hints; since Spark SQL does not natively support key annotations, we added them as a separate extension. We implemented the row-level conditional-exchange primitive in

Q1	`Count(GroupBy(Join(A, B, "x" == "y"), "x"))`

Q5	`Count(GroupBy(Filter(`
	`  Join(A, B, "w"),`
	`  Contains("A.x", "xyz")`
	`  && ("B.x" + "B.y" > 10)`
	`  && ("A.y" > "B.y")), "x"))`

Table 2.2: DJoin queries.

SGX using the x86 conditional move instructions on data in registers, for which we assume that the attacker cannot see accesses to registers inside enclaves. Federated query execution is coordinated by the querying JVM, which maintains a connection to each cluster in the federation using Spark's remote query functionality via the SparkSession.

2.8 Evaluation

In this section we evaluate OCQ's performance against outsourced and multi-party computation. We measure the overhead of OCQ's security guarantees. We explore an alternative design where SGX-enabled machines are required at only one site, and show that requiring SGX at all sites provides a significant speedup. We explore the speedup of our query planner compared to a traditional query planner unaware of security. Finally, we evaluate the benefit of schema-aware padding versus the conventional filter push-up approach.

2.8.1 Setup

We performed benchmarks across 5 parties located in AWS us-east-1, AWS us-west-1, AWS eu-west-1, and AWS ap-northeast-1, and in our organization. Each party has approximately 10 MB/s bandwidth to each other party. Our organization's site has an SGX cluster with 5 machines, while the AWS sites use 5-node `r5.large` clusters with Intel's SGX simulation driver. We use a federated query workload derived from previous papers on federated analytics. From SMCQL [15] we use the comorbidity and aspirin count queries described in Section 2.5.3. From DJoin [103] we use queries 1–5. The DJoin queries are listed in Table 2.2. We generated synthetic table data with the following total size per table:

1. `diagnosis` - 1,024,000 rows, 10 GB
2. `medication` - 142,972 rows, 4.3 MB
3. DJoin `A` - 15,000 rows, 15 MB
4. DJoin `B` - 15,000 rows, 15 MB

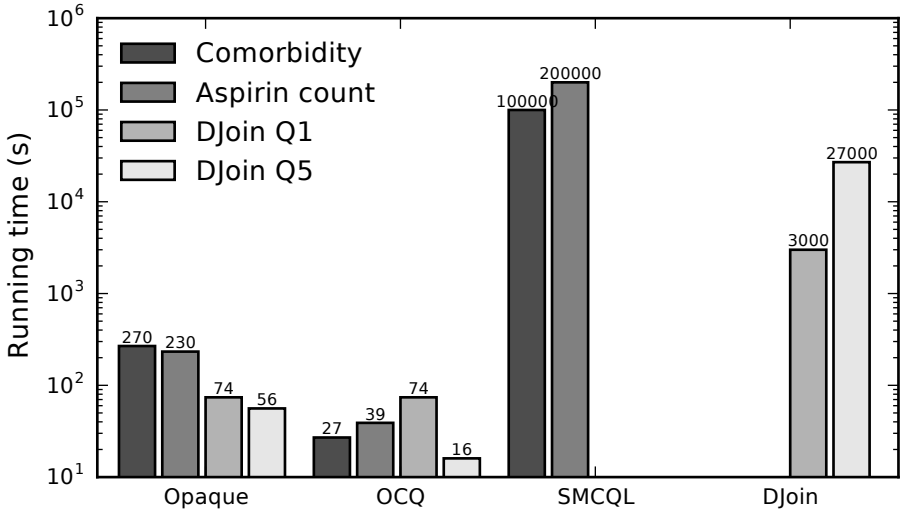


Figure 2.6: OCQ vs. competing systems. OCQ is orders of magnitude faster than SMCQL and DJoin due to its use of trusted hardware, and is faster than Opaque for most queries because it can execute initial filters in plaintext.

2.8.2 Comparison to other systems

Figure 2.6 compares OCQ to Opaque (outsourced computation) as well as SMCQL and DJoin (secure multi-party computation). For the first, since Opaque's implementation assumes at least 2MB of non-observable memory which speeds up its oblivious protocols considerably, we ran OCQ with the same configuration. For the second, we report the numbers from the SMCQL and DJoin papers because they are too slow to run on our dataset or not open source. They are also insufficient for the coopetitive setting (as explained in §2.9.1). OCQ is 1–4 orders of magnitude faster than SMCQL and DJoin due to its use of trusted hardware. Meanwhile, OCQ gains a performance advantage over Opaque, which also uses trusted hardware, for queries that begin with a substantially selective filter operation. Because the initial filter requires no communication between parties, it can be executed in plaintext at each party. The cardinality of the input to subsequent oblivious operators is then greatly reduced. For our synthetic data, the initial *c. diff* filter for the comorbidity query has $\approx 1\%$ selectivity, as does the result of the selection and join to find patients with heart disease who were prescribed aspirin for the aspirin count query. For OCQ we specified padding bounds that reflect this selectivity, because we do not wish to treat the selectivity parameter as sensitive. In contrast, Opaque must first perform an oblivious filter over each full relation. Because intermediate relation sizes within our query workload tend to shrink as the query progresses, this initial oblivious filter tends to dominate the running time.

Our reported query times include network transfer time, including the time required to transfer the full relations to the cloud for each Opaque query. Figure 2.7 shows the proportion of query time spent in network transfer. These transfers occur in parallel, so 5-node cluster uses its full aggregate

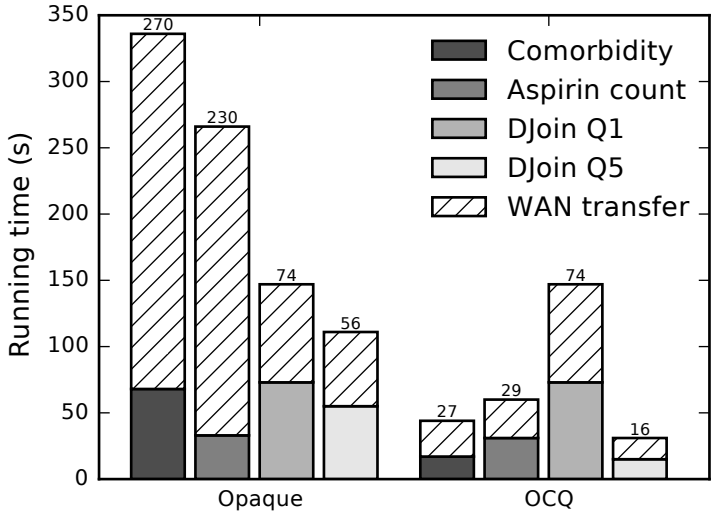


Figure 2.7: OCQ vs. Opaque, highlighting network transfer time. OCQ retains an advantage even assuming an infinitely fast network because it can execute initial filters in plaintext rather than using oblivious operators on the full inputs.

bandwidth to transfer each relation. For the medical queries, this transfer is the dominant factor in Opaque's query times due to the large data sizes involved. Although the uploaded data could be reused across queries, we include it in the query times for consistency with the other systems. Data reuse is not always possible, for example in case of frequently-changing data.

We also evaluated OCQ without any non-observable memory assumption (namely, the attacker can observe any party table data in any part of memory) to show that its performance remains much better than MPC-based systems. We compared OCQ against AgMPC [136], a state-of-the-art maliciously secure MPC framework (§2.9.1). We ran a query consisting of a referential integrity inner equi-join on two equal-sized synthetic tables, each containing two 32-bit integers. The first integer in each table was the join key. Figure 2.8 shows that OCQ is up to 219× faster than AgMPC on this query.

2.8.3 Overhead of security

Figure 2.9 compares OCQ to alternative uses of Opaque with varying security guarantees to show the overhead of OCQ's security. "Outsourced Opaque" is as before. "Plaintext federated" refers to an alternative federated configuration where all computation runs in plain text rather than within SGX. This configuration might be suitable for a network of non-competing entities. "Outsourced Spark SQL" refers to a configuration similar to Opaque but where computation is run in plaintext rather than in SGX. This option provides no security guarantees, as a server-side attacker could access the data in full.

Figure 2.9 shows that OCQ introduces 2.2–12× overhead compared to a plaintext federated

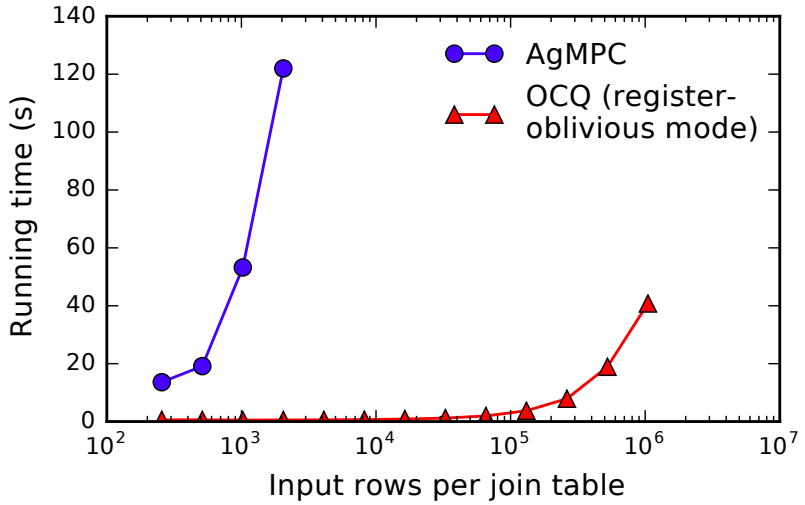


Figure 2.8: Performance of register-oblivious OCQ versus AgMPC. OCQ provides the same level of memory access pattern protection as AgMPC, but scales much better and is up to 219x faster.

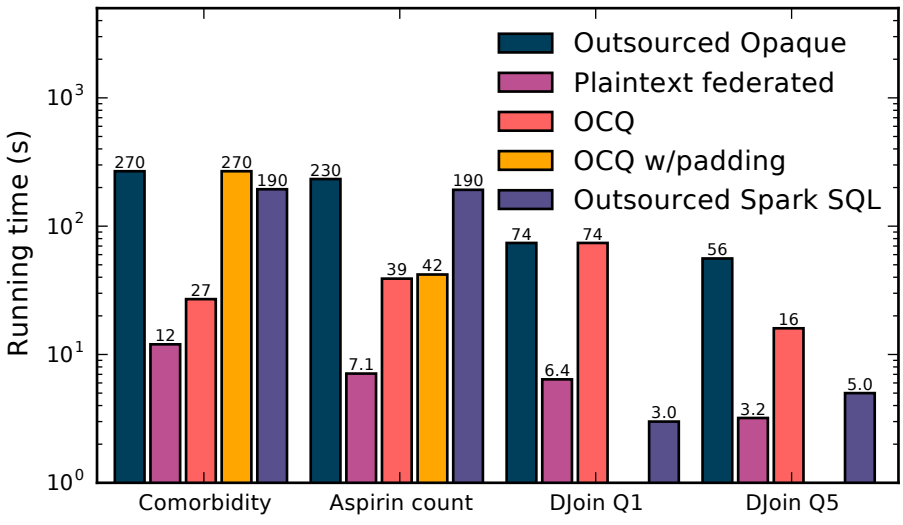


Figure 2.9: Overhead of OCQ's security. OCQ incurs 2.2–25× overhead compared to the federated and outsourced Spark SQL baselines, which provide no security.

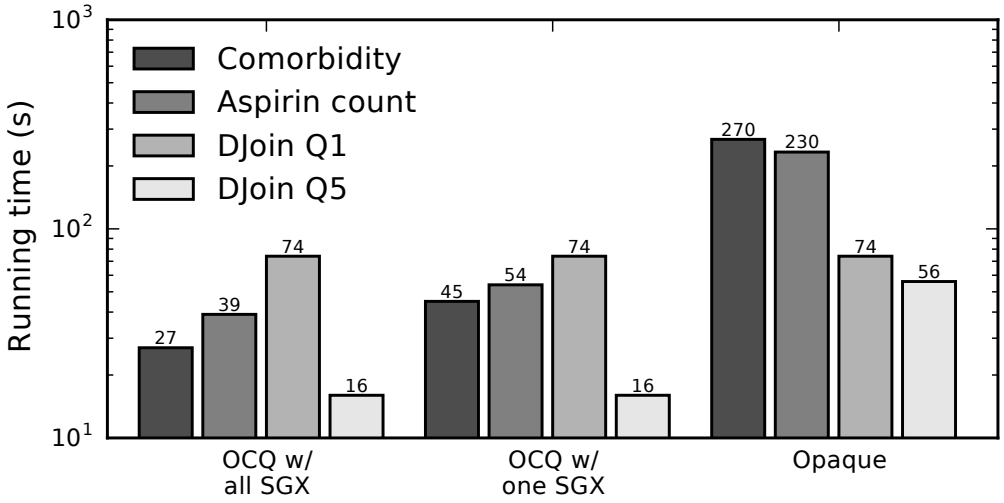


Figure 2.10: Availability of SGX enclaves at all sites provides 1.3–1.6× speedup compared to having SGX at only one site.

configuration due to its use of oblivious operators. Additionally, outsourced Spark SQL outperforms OCQ by up to 25× for the DJoin queries, which use small relations without any initial filters, and whose complexity is entirely in the core join computation, which is expensive to perform obliviously. However, OCQ outperforms the outsourced configurations for the medical queries, which do contain initial filters. Note that as before, we include network transfer time in query time, including the time to upload full tables for the outsourced configurations.

2.8.4 Benefit of having SGX at each site

We next explore the design choice in OCQ that each site must have its own local SGX cluster. This represents a constraint to its adoption, since SGX deployments are not widespread. However, we observe that some queries significantly benefit from this choice. Figure 2.10 compares OCQ to an alternative design ("OCQ w/ one SGX") where operations on multiple parties' data such as joins and aggregations require the data to be collected to a single SGX cluster first. We observe a 1.3–1.6× speedup over the alternative design for the medical queries because the use of broadcast joins allows the components of the broadcast join, namely an oblivious join at each party, to operate on less data. Since all oblivious joins run in parallel, this results in a speedup for the initial join, which dominates the query plans.

2.8.5 Benefit of schema-aware padding

We compare the performance of our schema-aware padding approach to a baseline query without padding and to an implementation of Opaque's "filter push-up" approach. In the latter, filters and

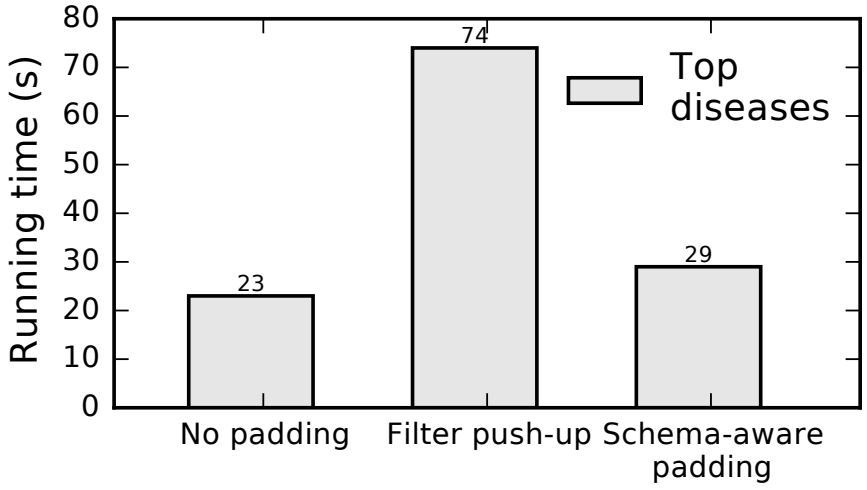


Figure 2.11: Schema-aware padding provides a 2.5× speedup compared to filter push-up, and is only 26% slower than the baseline plan without padding.

aggregations on sensitive relations always return one output row for each input row, with rows that did not match the predicate or rows other than the first one in each group resulting in a dummy output row. All dummy rows are filtered out at once as the final step of the query, thus hiding intermediate result sizes. This approach results in large costs when intermediate output is transferred over the network.

With appropriate sensitivity step-down hints for the medical dataset, schema-aware padding on the aspirin count and comorbidity queries results in the same plan as a suitably designed filter push-up approach. We therefore introduce a plausible new query on the medical dataset that finds the most costly diseases by grouping the `diagnosis` table on disease id and computing the total cost for each disease:

```
diagnosis.groupBy("disease")
  .agg(sum("cost") as "total_cost")
  .sort("total_cost").select("disease").take(5)
```

This query results in the physical plan without padding:

```
OblLimit 5
  OblProject [disease_id]
    OblSort [total_cost]
      OblAgg groupby[disease_id]
             agg[sum(cost_partial_sum) as total_cost]
        OblCollect
          FedAgg groupby[disease_id]
                 agg[sum(cost) as cost_partial_sum]
            FedEncScan diagnosis
```

Though the top-level aggregation occurs within SGX because its input is sensitive, this plan may leak the cardinality of the partial aggregates through SGX side channels. Our reimplementation of filter push-up results in the following:

```
OblLimit 5
  OblProject [disease_id]
    OblSort [total_cost]
      OblAggPadded groupby[disease_id]
                   agg[sum(cost_partial_sum) as total_cost]
        OblCollect
          FedPad diagnosis.cardinality
            FedAgg groupby[disease]
                   agg[sum(cost) as cost_partial_sum]
              FedEncScan diagnosis
```

This plan pads both levels of aggregation to the input size, hiding their cardinalities, but resulting in two costly oblivious sort operations on padded data. In contrast, our schema-aware padding recognizes that the cardinalities of the partial and final aggregates are bounded by the number of known disease codes, which is much smaller than the cardinality of the input `diagnosis` table and is public knowledge. It generates a nearly identical plan to filter push-up, with the difference that both aggregation operations pad to `disease.cardinality` instead of `diagnosis.cardinality`. Figure 2.11 compares the performance of all three plans using the input described in Section 2.8.1. The lower padding bound and consequently reduced oblivious sort cardinality give our schema-aware padding approach a 2.5× speedup compared to the filter push-up approach, and it is only 26% slower than the baseline plan without padding.

2.9 Related work

2.9.1 Cryptographic approaches

SMCQL [15] and Conclave [134] use secure multi-party computation (instead of hardware enclaves) to achieve federated analytics queries. Unlike OCQ, SMCQL and Conclave do not protect against a malicious attacker (the attacker is semi-honest), and their implementation is for only two or three parties. Further, their join scheme relies on joining non-sensitive attributes, unlike OCQ, which can join on sensitive data.

AgMPC [136] is likely the most relevant cryptographic framework to OCQ because it is n-party, maliciously-secure, and supports generic computation. As we show in §2.8, though, AgMPC is orders of magnitude slower than OCQ.

DJoin [103] and private intersection-sum [71] use multi-party computation to provide certain SQL operators. DJoin supports only count queries over equi-joins, while private intersection-sum supports sum queries over set intersections. Both assume a passive attacker and incur high overhead.

UnLynx [54] and MedCo [118] use partially-homomorphic encryption to provide filter-aggregate queries over multiple parties' data and are secure against malicious queriers. Compared to these systems, OCQ offers a greater range of functionality, but requires trust in hardware enclaves.

Cryptographic approaches have also been used for coopetitive machine learning training and prediction, which often involve secure aggregation [38, 19, 105, 64, 56, 100]. These approaches do not offer general analytics, and they largely assume a passive attacker or two non-colluding servers.

Encrypted databases such as CryptDB [114], AlwaysEncrypted [97], and Seabed [112] perform queries over encrypted data, but are not suited for the coopetitive setting.

2.9.2 Hardware enclave approaches

Systems for single-machine or distributed computation using hardware enclaves include SCONE [12], Graphene [32], Ryoan [69], Haven [17], VC3 [121], Cipherbase [9], and Opaque [145]. These systems assume a setting where all data is controlled by one party and are not designed for the coopetitive setting. Prochlo [18] offers privacy-preserving outsourced computation over many users' data using hardware enclaves. However, it requires centralizing the data, which encounters regulatory and logistical challenges in the coopetitive setting. Ohrimenko et al. [107] provides oblivious machine learning in SGX, but does not consider analytics in the federated setting and query planning. Oblix [98] focuses on oblivious point queries and does not support analytics and the decentralized setting.

2.9.3 Unencrypted federated databases

Collaborative query planning (CQP) [144] is a proposal for decentralized query planning in a multi-party setting where information sharing policy restricts centralized planning. Queries are instead broken into subqueries, independently planned by each party, and reassembled into a federated plan. CQP shares a setting with OCQ and complements it in the case where query planning information such as statistics must be treated as sensitive. Preference-aware query optimization (PAQO) [53, 52] is a proposal to extend SQL with users' declarations for where their data should be processed in a distributed database, with applications including restricting certain data from untrusted servers. PAQO allows the user to treat these declarations, called intensional descriptions [52], as preferences to be optimized rather than hard constraints. In future work, a similar approach could be applied to OCQ.

2.9.4 Differential privacy

A complementary and synergetic direction to OCQ are differential-privacy systems like Flex [76] and Chorus [77], which offer differential privacy for SQL queries via query rewriting. The queries they produce can be used as input to OCQ. Hence, one could add differential privacy to a query result before sharing it among the parties in OCQ. Shrinkwrap [16] provides differential privacy specifically for federations, and can be used with OCQ to reduce the amount of padding for intermediate results.

2.10 Summary

In this chapter we proposed OCQ, an efficient framework for oblivious coopetitive analytics using hardware enclaves. OCQ's contributions are its query planner design, which supports flexible party-specific sensitivity rules, its mechanism for propagating and refining padding upper bounds based on foreign key constraints, and its mixed-sensitivity algorithms.

Chapter 3

GraphFrames: An Integrated API for Mixing Graph and Relational Queries

We now turn to the topic of complex analytics, particularly graph analytics and querying. This chapter builds upon GraphX [59], a framework I coauthored that enables large-scale graph processing within Apache Spark rather than requiring a separate, specialized engine. In this chapter we study how to generalize the ideas behind GraphX to support pattern matching as well as graph algorithms.

3.1 Introduction

Analyzing the graphs of relationships that occur in modern datasets is increasingly important, in domains including commerce, social networks, and medicine [93, 28, 13]. To date, this analysis has been done through specialized systems like Neo4J [137], Titan [132] and GraphLab [89]. These systems offer two main capabilities: *pattern matching* to find subgraphs of interest [137, 65, 51] and *graph algorithms* such as shortest paths and PageRank [91, 89, 59].

While graph analytics is powerful, running it in a separate system is both onerous and inefficient. Most workflows involve building a graph from existing data, likely in a relational format, then running search or graph algorithms on it, and then performing further computations on the result. With isolated graph analysis systems, users have to move data manually and there is no optimization of computation across these phases of the workflow. Several recent systems have started to bridge this gap by running graph algorithms on a relational engine [59, 75, 50], but they have no support for pattern matching and do not optimize across graph and relational queries.

We present GraphFrames, an integrated system that can combine relational processing, pattern matching and graph algorithms and optimize computations across them. GraphFrames generalize the ideas behind GraphX [59] and Vertexica [75] by maintaining arbitrary *views* of a graph (e.g., triplets or triangles) and executing queries using joins across them. They then optimize execution across the relational and graph portions of the computation. A key challenge in achieving this goal is query planning. For this purpose, we extend a graph-aware dynamic programming algorithm by Huang et al. [67] to select among multiple input views and compute a join plan. We also propose an algorithm

```
gf = GraphFrame(vertices, edges)

triples = gf.pattern(
  "(x:User)->(p:Product)<-(y:User)")

pairs.where(pairs.p.category == "Books")
     .groupBy(pairs.p.name)
     .count()
```

Listing 3.1: An example of the GraphFrames API. We create a GraphFrame from two tables of vertices and edges, and then we search for all instances *pattern*, namely two users that bought the same product. The result of this search is another table that we can then perform filtering and aggregation on. The system will optimize across these steps, e.g., pushing the filter above the pattern search.

for suggesting new views based on the query workload.

To make complete graph analytics workflows easy to write, GraphFrames provide a declarative API similar to "data frames" in R, Python and Spark [117, 111, 11] that integrates into procedural languages like Python. Users build up a computation out of relational operators, pattern matching, and calls to algorithms, as shown in Listing 3.1. The system then optimizes across these steps, selecting join plans and performing algebraic optimizations. Similar to systems like Pig [110] and Spark SQL [11], the API makes it easy to build a computation incrementally while receiving the full benefits of relational optimization. Finally, the GraphFrames API is also designed to be used *interactively*: users can launch a session, define views that will aid their queries, and query data interactively from a Python shell. Unlike current tools, GraphFrames let analysts perform their complete workflow in a single system.

We have implemented GraphFrames over Spark SQL [11], and made them compatible with Spark's existing DataFrame API. We show that GraphFrames match the performance of other distributed graph engines for various tasks, while enabling optimizations across the tasks that would not happen in other systems. Support for multiple views of the graph adds significant benefits: for example, materializing a few simple views can speed up queries by 10× over the algorithm in [67]. In addition, viewing graph data as relations makes it easy to write domain-specific optimizations such as attribute-based partitioning [147]. Finally, by building on Spark, GraphFrames interoperate easily with custom UDFs (e.g., ETL code) and with Spark's machine learning library and external data sources.

In summary, our contributions are:

- A declarative API that lets users combine relational processing, pattern matching and graph algorithms into complex workflows and optimizes across them.
- An execution strategy that generalizes those in Vertexica and GraphX to support multiple views of the graph.
- A graph-aware query optimization algorithm that selects join plans based on the available views.

- An implementation and evaluation of GraphFrames on Spark.

3.2 Motivation

In this section, we highlight several use cases that require the mixing of relational processing, graph pattern matching and iterative graph algorithms. We then argue that current systems cannot address these use cases efficiently. We discuss how GraphFrames, the first unified graph analytics system, can fix these problems.

3.2.1 Use Cases

E-Commerce One very important problem E-Commerce retailers need to solve is product recommendation. One representative system implementation is shown in Figure 3.1.

Since user activities are captured and stored in different systems such as transactional databases for purchase and web logs for click information and document databases or other NoSQL databases for product review data. The first step is to perform Extract-Transform-Load (ETL) to extract information on users, products, ratings and purchasing information from these relevant systems. The output is typically written in a distributed file system such as HDFS.

The second step typically involves running collaborative filtering to compute predicted ratings of users, i.e. to uncover latent ratings not present in the dataset. This is done for all users. Collabrative filtering computation are ususally implemented as iterative graph algorithms in a graph parallel processing engine such as GraphX, Apache Giraph. The output is also usually written in a distributed file system or database for further processing.

Retailers typically do not rely on the results from the second step. They would like to further customize their recommendation considering group behavior to perform more personalized recommendation. This step involves finding user activity patterns to refine or rank recommendation results. This step is well-suited for graph pattern matching. Relational processing after this step can also be carried out, e.g. aggregation and filtering, joining with other datasets.

As shown in Figure 3.1, the whole pipeline crosses multiple systems, and data gets copied into and out of the storage system multiple times. Each system has their own API and data structures.

Economic Graph LinkedIn is in the process of constructing the economic graph. The economic graph can help connect employees and employers much better. For example, company A's recruiter might want to find employees who knows Golang and worked at Google. It then filter the list with those who are connected to at least one employee at company A in three hops. LinkedIn might perform candidate recommendations for all its supported companies.

Instagram Follow Graph Instagram suggests users who else they may want to follow. This involves collaborative filtering to identify similar accounts who has already followed some other users. This is currently performed in Apache Giraph. The results are then further processed for

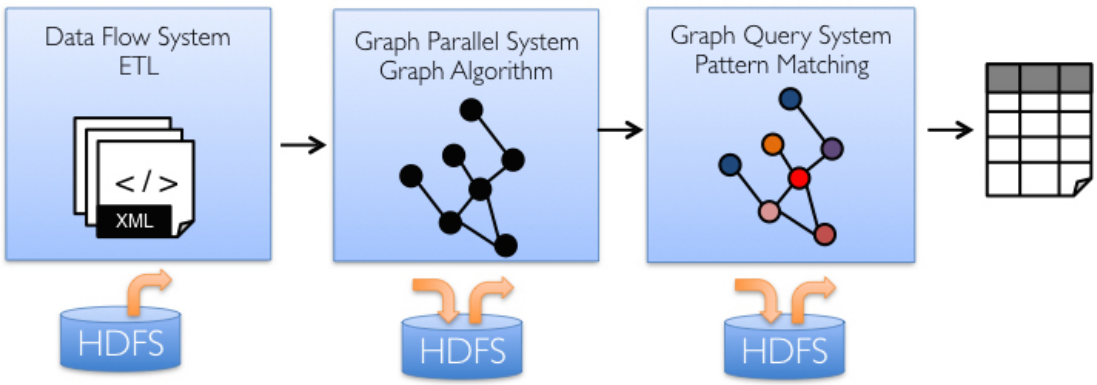


Figure 3.1: Graph Analytics Pipeline

account recommendation. This involves ranking other sources using boosted tree [28]. GraphFrames can perform ETL, collaborative filtering, and boosted tree processing in a single system.

Genomics In genomics, scientists would like to find some patterns. For example, find clones with name YWXD1000 and type YAC [13]. They then may perform clustering on the found patterns. GraphFrames can easily handle this use case which combines graph pattern matching and clustering.

3.2.2 Challenges and Requirements

All the above use cases require a combination of Extract-Transform-Load (ETL), iterative graph algorithms, and graph pattern matching. For example, to find users with similar interests in Amazon, we need to do ETL to extract the graph. We then perform collaborative filtering to fill in the missing edges between users and products. With the user-product rating graph, we can use pattern matching to find users who cobought similar products. For computation in the economic graph, we need graph pattern matching to find related candidates. We can run iterative graph algorithms to identify community of users of similiar skill sets. Instagram suggests users to follow by extracting the follow graph in an ETL step, running collaborative filtering using Apache Giraph, and ranking potential accounts to suggest using a separate machine learning system. For genomics, we need graph pattern matching and iterative graph algorithm to perform clustering.

However, there is currently no single system capable of efficiently executing all three computations: relational processing or dataflow processing, graph pattern matching, and iterative graph algorithms. As shown in Figure 3.1, a typical graph analytics pipeline may begin by using a dataflow system such as Apache Hadoop MapReduce or Apache Spark to load a dataset from HDFS and extract the graph. Once the graph is extracted, it may be written back into HDFS. Then Apache Giraph or GraphX will process the graph and output the results. For the Amazon use case, graph pattern matching is then performed to compute users who bought similar products. The analytics pipeline can be repeated many times during exploratory studies.

There are three major drawbacks of the current implementation.

- The graph analytics pipeline crosses different systems multiple times. Intermediate results are written into a distributed file system. This has serious performance overheads. It can not keep up with interative exploratory studies.
- No end-to-end optimization can be carried out and no sharing of intermediate state and data structures.
- Different systems have different APIs. There is no single abstraction to ease system development.

3.2.3 Need for a Unified System

To support graph analytics in batch and interactive model, the system needs to be responsive and support a unified programming model. Indeally, we would like a unified system to perform ETL, iterative graph algorithms and graph pattern matching.

GraphFrames provides the ability to stay within a single framework throughout the analytics process. This removes the need to learn and support mutiple systems (e.g., Figure 3.1) and plumbing to move between systems. As a result, it is substantially easier to interactively transform, and compute on large graphs and share data-structures across stages of the pipeline. GraphFrames support the above-mentioned use case very well. GraphFrames built on top of Spark DataFrames can perform powerful ETL processing, e.g. to construct the economic graph. GraphFrames support a concise and declarative API for graph pattern matching and iterative graph algorithms. It is languaged integrated with Scala. This makes it easy to use language features to create user defined functions (UDFs).

3.3 GraphFrame API

The main programming abstraction in GraphFrames' API is a *GraphFrame*. Conceptually, it contains of two relations (tables) representing the vertices and edges of the graph, as well as a set of materialized views of subgraphs. The vertices and edges may have multiple *attributes* that are used in queries. The views are defined using *patterns* to match various shapes of subgraphs, as we shall describe in Section 3.3.2. For example, a user might create a view of all the triangles in the graph, which can then be used to quickly answer other queries involving triangles.

GraphFrames expose a concise language-integrated API that unifies graph analytics and relational queries. We based this API on DataFrames, a common abstraction for data science in Python and R [111, 117] that is also available as a declarative API on Spark SQL [11]. In this section, we first cover some background on DataFrames, and then discuss the additional operations available on GraphFrames. We demonstrate the generality of GraphFrames for analytics by mapping the core primitives in GraphX into GraphFrame operations. Finally, we discuss how GraphFrames integrate with the rest of Apache Spark (e.g., the machine learning library).

All the code examples are shown in Python. We show the GraphFrame API itself in Scala because it explicitly lists data types.

3.3.1 DataFrame Background

DataFrames are the main programming abstraction for manipulating tables of structured data in R, Python, and Spark. Different variants of DataFrames have slightly different semantics. For the purpose of this chapter, we describe Spark's DataFrame implementation, which we build on [11]. Each DataFrame contains data grouped into named columns, and keeps track of its own schema. A DataFrame is equivalent to a table in a relational database, and can be transformed into new DataFrames using various relational operators available in the API.

As an example, the following code snippet computes the number of female employees in each department by performing aggregation and join between two data frames:

```
employees
  .join(dept, employees.deptId == dept.id)
  .where(employees.gender == "female")
  .groupBy(dept.id, dept.name)
  .agg(count("name"))
```

`employees` is a DataFrame, and `employees.deptId` is an expression representing the `deptId` column. Expression objects have many operators that return new expressions, including the usual comparison operators (e.g., == for equality test, > for greater than) and arithmetic ones (+, -, etc). They also support aggregates, such as `count("name")`.

Internally, a DataFrame object represents a *logical plan* to compute a dataset. A DataFrame does not need to be materialized, until the user calls a special "output operation" such as `save`. This enables rich optimization across all operations that were used to build the DataFrame.[1]

In terms of data type support, DataFrame columns support all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions. Complex data types can also be nested together to create more powerful types. In addition, DataFrame also supports user-defined types [11].

3.3.2 GraphFrame Data Model

A GraphFrame is logically represented as two DataFrames: an edge DataFrame and a vertex DataFrame. That is to say, edges and vertices are represented in separate DataFrames, and each of them can contain attributes that are part of the supported types. Take a social network graph for an example. The vertices can contain attributes including name (string), age (integer), and geographic location (a struct consisting of two floating point values for longitude and latitude), while the edges can contain an attribute about the time a user friended another (timestamp). The GraphFrame model supports user-defined attributes with each vertex and edges, and thus is equivalent to the property

[1] This aspect of Spark DataFrames is different from R and Python; in those languages, DataFrame contents are materialized eagerly after each operation, which precludes optimization across the whole logical plan [11].

graph model used in many graph systems including GraphX and GraphLab. GraphFrame is more general than Pregel/Giraph since GraphFrame supports user-defined attributes on edges.

Similar to DataFrames, a GraphFrame object is internally represented as a logical plan, and as a result the declaration of a GraphFrame object does not necessarily imply the materialization of its data.

Next, we explain how a GraphFrame can be constructed and operations available on them.

```
class GraphFrame {
  // Different views on the graph
  def vertices: DataFrame
  def edges: DataFrame
  def triplets: DataFrame
  // Pattern matching
  def pattern(pattern: String): DataFrame

  // Relational-like operators
  def filter(predicate: Column): GraphFrame
  def select(cols: Column*): GraphFrame
  def joinV(v: DataFrame, predicate: Column)
    : GraphFrame
  def joinE(e: DataFrame, predicate: Column)
    : GraphFrame

  // View creation
  def createView(pattern: String): DataFrame

 // Partition function
 def partitionBy(Column*) GraphFrame
}
```

Listing 3.2: GraphFrame API in Scala

Graph Construction

A GraphFrame can be constructed using two DataFrames: a vertex DataFrame and an edge DataFrame. A DataFrame is merely a logical view (plan) and can support a wide range of sources that implement a data source API. Some examples of a DataFrame input include:

- a table Spark SQL's system catalog
- a table in an external relational database through JDBC
- JSON, Parquet, Avro, CSV files on disk
- a table in memory in columnar format

- a set of documents in ElasticSearch or Solr
- results from relational transformations on the above

The following code demonstrates constructing a graph using a user table in a live transactional database and the edges table from some JSON based log files in Amazon S3:

```
users = read.jdbc("mysql://...")
likes = read.json("s3://...")
graph = GraphFrame(users, likes)
```

Again, since DataFrames and GraphFrames are logical abstractions, the above code does not imply that `users`, `likes`, or `graph` are materialized.

Edges, Vertices, Triplets, and Patterns

A GraphFrame exposes four tabular views of a graph: edges, vertices, triplets, and a pattern view that supports specifying graph patterns using a syntax similar to the Cypher pattern language in Neo4J [137].

The edges view and the vertices view should be self-evident. The triplets view consists of each edge and its corresponding source and destination vertex attributes. It can actually be constructed using the following 3-way join:

```
e.join(v, v.id == e.srcId)
 .join(v, v.id == e.dstId)
```

We provide it directly since the triplets view is used commonly enough. Note that edges, vertices, and triplets views are also the three fundamental views in GraphX, and GraphFrames is at least as expressive as GraphX from the perspective of views.

In addition to the three basic tabular views, a GraphFrame also supports a pattern operator that accepts a graph pattern in a Cypher-like syntax and returns a DataFrame consisting of edges and vertices specified by the pattern. This pattern operator enables easy expression of pattern matching in graphs.

Typical graph patterns consist of two nodes connected by a directed edge relationship, which is represented in the format *()-[]->()*. Nodes are specified using parentheses (), and relationships are specified using square brackets, []. Nodes and relationships are linked using an arrow-like syntax to express edge direction. The same node may be referenced in multiple relationships, allowing relationships to be composed into complex patterns. Additionally, nodes and edges can be constrained using inline type predicates expressed using colons.

For example, the following snippet shows a user `u` who viewed both item `x` and item `y`.

```
(u:Person)-[viewed]->(x: Item), u-[viewed]->(y: Item)
```

The resulting DataFrame from the above pattern should contain 3 structs: `u`, `x`, and `y`.

Note that the pattern operator is a simple and intuitive way to specify pattern matching. Under the hood it is implemented using the join and filter operators available on a GraphFrame. We provide it because it is often more natural to reason about graphs using patterns than using relational

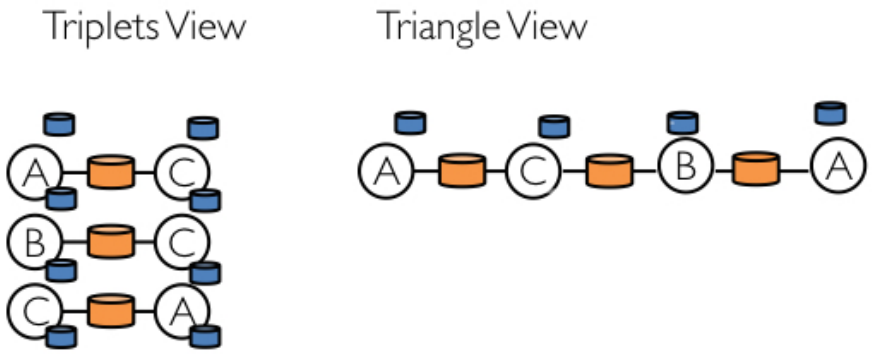


Figure 3.2: View Reuse

joins. The pattern operator can also be combined with other operators, as demonstrated in the next subsection.

Programatic Pattern Generation and Pattern Library Patterns such as cliques can be cumbersome to specify for interactive queries. We therefore provide a pattern library to programmatically generate and compose patterns. For example, a star of size K can be specified as `(hub, spokes) = star(nodePred, edgePred, K)`. `nodePred` and `edgePred` filters out nodes and edges. The pattern returns a hub and a list of K-1 spokes. Their names can then be used in further pattern specification, or materialized immediately.

View Creation

To enable reuse of computation, GraphFrames support view creation. The system materializes the view internally and uses it to speed up subsequent queries. For example, in Figure 3.2, if we create a view on triplets, we can reuse it to create a triangle view. The system will avoid rematerializing the triplet view for computing the triangle view. We will discuss how our query planner performs view selection to optimize the computation in the next section.

Relational Operators

Since a GraphFrame exposes the four tabular views, it already supports all the relational operators (e.g. project, filter, join) available on these tabular views. Relational operators on these views can already support many graph analysis algorithms. For example, the following snippet computes the out-degree for each vertex:

```
g.edges.groupBy(g.edges.srcId).count()
```

In addition to relational operators on the tabular views, a GraphFrame also includes a few relational-like operators on a graph:

Project The `select` operator projects the relevant columns of the corresponding vertices DataFrame and edges DataFrame. The result is a graph with a subset of the vertex attributes

and edge attributes. For example, *gf1 = gf.select(gf.vertices.age, gf.edges.likes)* projects only the `age` attribute on vertices and `likes` attribute on edges. The select operator also supports all the expressions available in DataFrames, including equality, arithmetics, string functions.

Filter The `filter` operator, as the name implies, filters the graph based on edge or vertex attributes and returns the subgraph matching the filter. If a node is filtered, all the edges associated with it will be filtered out, and vice versa.

Join The `joinV` operator creates a new graph by joining the existing graph's vertices with a different DataFrame and produces a new graph with more attributes for each vertex. It supports all common join types such as inner join, left outer, right outer, full outer, and semi. It has two common use cases. The first is to enhance a graph by filling in more attributes. For example, the following snippet enhances an existing social graph by adding user interest information:

```
g.joinV(interests,
  g.vertices.userId == interests.userId,
  "right_outer")
```

The second use case is to combine `joinV` with `select` to model the Apply-Scatter phase in the Gather-Apply-Scatter (GAS) abstraction in GraphLab. This is similar to GraphX's `leftJoinV` operator, except the GraphFrame join is more general because it can join on arbitrary attributes.

Similarly, the `joinE` operator creates a new graph by joining the existing graph's edges with a different DataFrame and produces a new graph with more attributes for each edge.

Attribute-Based Partitioning

Similar to GraphX, GraphFrames by default partitions a graph based on the natural partitioning scheme of the edges. In [59], it was shown that natural partitioning can lead to great performance when the input is pre-partitioned.

In addition, GraphX supports partitioning a graph based on arbitrary vertex or edge attributes. This is more general than GraphX or Giraph because they only support partitioning on vertex identifiers. This enables users to partition a graph based on their domain-specific knowledge that can lead to strong data locality and minimize data communications.

Take the Amazon dataset [93] for example. The following snippet partitions the bipartite graph based on product categories:

```
g.partitionBy(g.vertices.productCategory)
```

Intuitively, customers are more likely to buy products in the same category. Partitioning the Amazon graph this way puts products of the same categories and their associated edges closer to each other. In Section 3.6.4, we demonstrate that this partitioning schema does lead to substantial reduction in data communication.

User-defined Functions

GraphFrame also supports arbitrary user-defined functions (UDFs) in Scala, Java, and Python. The `udf` function accepts a lambda function as input and creates an expression that can be used in relational and graph projection. For example, given a `model` object for a machine learning model, we could create a UDF predicting some user behavior based on users' `age` and `registrationTime` attributes.

```
model: LogisticRegressionModel = ...
predict = udf(lambda x, y: model.predict(Vector(x, y)))
g.select(
  predict(g.vertices.age, g.vertices.registrationTime))
```

Unlike database systems which often require UDFs to be defined in a separate programming environment that is different from the primary query interfaces, our GraphFrame API supports inline definition of UDFs. We do not need complicated packaging and registration process found in other database systems.

3.3.3 Generality of GraphFrames

As simple as it is, the GraphFrame abstraction is powerful enough to express many workloads. We demonstrate the expressiveness by mapping all GraphX operators to operators in GraphFrame. Since GraphX can be used to model the programing abstractions in GraphLab, Pregel, and BSP [59], by mapping GraphX operations to GraphFrame, we demonstrate that GraphFrame is at least as expressive as GraphX, GraphLab, Pregel, and BSP.

GraphX's operators can be divided into three buckets: collection views, relational-like operators, and graph-parallel computations. Section 3.3.2 already demonstrated that GraphFrame provides all the three fundamental collection views in GraphX (edges, vertices, and tripets).

All relational-like operators in GraphX can be trivially mapped one-to-one to GraphFrame operators. For example, the `select` operator is a superset of GraphX's `mapV` and `mapE` operators, and `joinV` and `joinE` are the generalized variant of GraphX's `leftJoinV` and `leftJoinE` operators. The `filter` operator is a more general version of GraphX's `subgraph` operator.

In GraphX, graph-parallel computations consist of `aggregateMessages`[2] and its variants. Similar to the Gather phase in the GAS abstraction, `aggregateMessages` encodes a two-stage process of graph-parallel computation. Logically, it is the composition of a projection followed by an aggregation on the triplets view. In [59], it was illustrated using the following SQL query:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```

This SQL query can indeed be expressed using the following GraphFrame operators:

```
g.triplets
 .select(mapF(g.triplets.attribute[s]).as("mapOutput"))
 .groupBy(g.triplets.dstId)
 .agg(reduceF("mapOutput"))
```

[2] `aggregateMessages` was called `mrTriplets` in [59], but renamed in the open source GraphX system.

We demonstrated that GraphFrame can support all the operators available in GraphX and consequently can support all operations in GraphLab, Pregel, and BSP. For convenience, we also provide similar APIs as GraphX's Pregel variant in GraphFrame for implementing iterative algorithms. We have also implemented common graph algorithms including connected components, PageRank, triangle counting.

3.3.4 Spark Integration

Because GraphFrames builds on top of Spark, this brings three benefits. First, GraphFrames can load data from and save data into existing Spark SQL data sources such as HDFS files in Json, Parquet format, HBASE, Cassandra, etc. Second, GraphFrame can use a growing list of machine learning algorithms in MLlib. Third, GraphFrames can call Spark DataFrame API. As an example, the following code reads user-product rating information from HDFS into a DataFrame. We then select the review text and use user ID and product ID pair as the key. We can call the topic model to learn the topics of reviews. With the topics, we can compute similar products, etc as in [93] and do graph pattern matching to uncover user communities who bought similar products.

```
corpus = rating.read.parquet("hdfs:///...")
 .select(pair(user_id, product_id), review_txt)
ldaModel = LDA.train(corpus, k=10000)
topics = ldaModel.topicsMatrix()
```

3.3.5 Putting It Together

We show that the ease of developing an end-to-end graph analytics pipeline with an example in Listing 3.3. As we discussed in Section 3.2, for ecommerce, we would like to group users of similar intersts.

The first step is to perform ETL to extract information on users, products, ratings and cobought. They are represented as DataFrames. We then construct a GraphFrame `graph`. The vertices contain both user nodes and product nodes. The edges are between users and products. An edge exists between a user and a product if the user rated the product. This is a bipartite graph.

For the second step, we run collaborative filtering to compute predicted ratings of users, i.e. to uncover latent ratings not present in the dataset. We then create a graph `densifiedGraph` with the same vertex node as `graph` and more edges by adding product-product edges. A product-product edge is added if the two are cobought.

As the final step, we will find users who have good ratings for at least two products together. Instead of finding pairs of users, we can find group of users of size K.

This example shows the ease of using the GraphFrames API. We performed ETL, iterative graph algorithms and graph pattern matching in one system. It is much more intuitve than coding the pipeline in SQL. Language integration also makes it easy to plug in UDFs. For example, we can create a UDF to extract product topics and topics user interested.

In the next section, we will highlight the opportunities for joint optimization.

```
# 1. ETL
# users: [id: int, attributes: MapType(user_name)]
# products: [id: int, attributes: MapType(brand,
# category, price)]
# ratings: [user_id: int, product_id: int,
# rating: int, review_text: string]
# cobought: [product_1_id: int, product_2_id: int]

vertices = users.union(products)
graph = GraphFrame(vertices, ratings)

# 2. Run ALS to get top 1M inferred recommendations
# predictedRatings: [user_id: int, product_id: int,
# predicted_rating: int]
predictedRatings = ALS.train(graph, iterations=20)
  .recommendForAll(1e6)

densifiedGraph = GraphFrame(vertices,
  ratings.union(predictedRatings).union(cobought))

# 3. Find groups of users with the same interests
densifiedGraph.pattern("""(u1)-[r1]->(p1);
  (u2)-[r2]->(p2); (p1)-[]->(p2)""")
  .filter("r1.rating > 3 && r2.rating > 3")
  .select("u1.id", "u2.id")
```

Listing 3.3: GraphFrame End-to-End Example in Python

3.4 Query Optimization

GraphFrame operators, including both the graph as well as the relational operators, are compiled to relational operators. Thereafter, we optimize the complete pipeline by extending Catalyst. To do this, the GraphFrame query planner extends the dynamic programming algorithm of Huang et al. [67] to the distributed setting, with two major changes:

View rewrite The user can register arbitrary materialized views and the planner will automatically rewrite the query to reuse a materialized view when appropriate. This is useful because pattern queries could be very expensive to run and reusing computations across several queries can improve the user experience. GraphFrame API also allows users to get suggestions for the views to create. Finally, we also describe how we can further extend the query planning to create the views adaptively.

Vertex cut/2D partitioning-aware planning The system partitions distributed graphs using vertex cuts, which are more communication-efficient than the more commonly-used edge cuts. The planner is aware of the vertex cut layout as well as a 2D partitioning heuristic which provides a communication bound, and it chooses join sites appropriately to minimize communication.

By building on top of Spark SQL, GraphFrames also benefit from whole-stage code generation.

3.4.1 Query Planner

The dynamic programming algorithm proposed in [67] recursively decomposes a pattern query into fragments, the smallest fragment being a set of co-partitioned edges, and builds a query plan in a bottom-up fashion. The original algorithm considers a single input graph. In this chapter, we extend it to views, i.e., the algorithm matches the views in addition to matching the pattern query. The input to the algorithm is the base graph G, a set of graph views $\{GV_1, GV_2, .., GV_n\}$, and the pattern query $Q = (V_q, E_q)$. Each graph view GV_i consists of the view query that was used to create the view and a cost estimator CE_i. The algorithm also takes the partitioning function P as an input, as opposed to a fixed partitioning in the original algorithm. The output is the best query plan (lowest cost) to process the query Q.

The algorithm starts by recursively decomposing the pattern query into smaller fragments and building the query plan for each fragment. At the leaf level, i.e., when the fragment consists of only a single edge, we lookup the edge (along with its associated predicates) in the base graph. At higher levels, we combine the child query plans to produce larger plans. At each level, we also check whether the query fragment matches with the view query of any of the graph views. In case a match is found, we add the view as a candidate solution to the query fragment. This also takes care of combining child query plans from multiple graph views, i.e., we consider all combinations of the graph views and later pick the best one. Algorithm 1 shows the extended algorithm for finding plans using views. Each time a new plan is generated for a query fragment, we match the fragment with the set of graph views, as shown in blue in Algorithm 1. Algorithm 2 shows the pseudocode for view matching. For every query plan solution, we check whether its query fragment is equivalent to a view query[3] and add the view to the solution set in case a match is found. Note that we keep both the solutions, one which uses the view and one which does not, and later pick the best one. Also note that we match the views on the logical query fragments in a bottom-up fashion, i.e., a view matched a lower levels could still be replaced by a larger view (and thus more useful view) at the higher levels.

Combining graph views, however, produces a new (intermediate) graph view and so we need to consider the new (combined) cost estimate when combining it further. To handle this, we keep track of four things in the query plan solution at each level: (i) the query plan, (ii) the estimated cost, (iii) the partitioning, and (iv) the cost estimator. When combining the solutions, we combine their cost estimates as well[4].

Figure 3.3 illustrates the query planning using views. The system takes the given pattern query and the three graph views, V_1, V_2, and V_3 as inputs. The linear decomposition (recursively) splits the query into two fragments, such that at least one of them is not decomposable (i.e., it is either a single edge or co-partitioned set of edges). The lower left part of Figure 3.3 shows one such linear decomposition and the corresponding candidate query plan using views. Here we match a view with a query fragment only when it contains the exact same set of edges. The bushy decomposition generates query fragments none of which may be non-decomposable (i.e., each query fragment

[3]Instead of looking for exact match, the algorithm could also be extended to match views which *contain* the query fragment, as in traditional view matching literature.

[4]We can do this more efficiently by pre-computing the estimates for all combinations of the graph views.

Algorithm 1: `FindPlanWithViews`

Input : query Q; graph G; views $GV_1, .., GV_n$; partitioning P
Output Solutions for running Q
:

```
1  if Q.sol != null then
2      return null; // already generated plans for Q
3  if Q has only one edge e = (v1, v2)) then
4      Q.sol = ("match e", scan cost of E_i, P(e_i), ce_i);
5      Q.sol = Q.sol ∪ MatchViews(Q.sol, views);
6      return;
7  if all edges in Q are co-partitioned w.r.t P then
8      Q.sol = ("co-l join of Q",co-l join cost i, P(Q), ce_i);
9      Q.sol = Q.sol ∪ MatchViews(Q.sol, views);
10     return;
11 T = φ ;
12 LD = LinearDecomposition(Q) ;
13 foreach linear decomposition (q1,q2) in LD do
14     FindPlan(q1);
15     FindPlan(q2 );
16     linearPlans = GenerateLinearPlans(q1,q2);
17     T = T ∪ linearPlans;
18     T = T ∪ MatchViews(linearPlans, views);
19 LDAGs = GenerateLayeredDAGs(Q) ;
20 foreach layered DAG d in LDAGs do
21     (q_1, q_2, ..., q_{n-1}, q_n) = BushyDecomposition(d) ;
22     for i from 1 to n do
23         FindPlan(q_i );
24     bushyPlans = GenerateBushyPlans(q_1, ..., q_n);
25     T = T ∪ bushyPlans;
26     T = T ∪ MatchViews(bushyPlans, views);
27 Q.sol = EliminateNonMinCosts(T);
```

Algorithm 2: `MatchViews`

Input : query plan solution set $\mathbb{S}$; graph views $GV_1, .., GV_n$
Output : query plan solution set from matching views

```
1 V = φ ;
2 foreach Solution S in 𝕊 do
3     foreach Graph View GV_i do
4         queryFragment = S.plan.query;
5         viewQuery = GV_i.query;
6         if viewQuery is equivalent to queryFragment then
7             V = V ∪ ("scan", scan cost of GV_i, GV_i.p, CE_i);
8 return V;
```

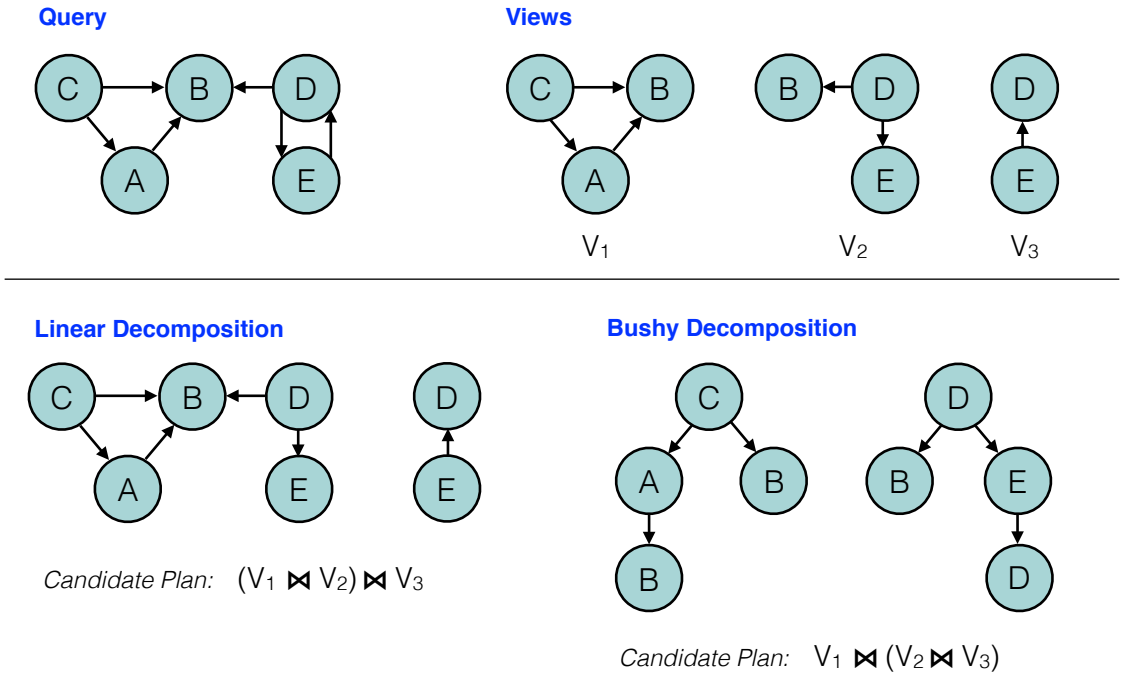


Figure 3.3: View Matching during Linear and Bushy Decompositions.

S.No.	View
1	$A \leftarrow B, A \leftarrow C, B \leftarrow F$
2	$A \leftarrow B, A \leftarrow C, C \leftarrow E$
3	$A \leftarrow B, A \leftarrow C, D \leftarrow B$
4	$A \leftarrow B, A \leftarrow C, E \leftarrow C$
5	$B \leftarrow A, B \leftarrow D, E \leftarrow B$

Table 3.1: Top-5 views of size three suggested by the system for the workload in Figure 3.5.

could be further split into linear or bushy decompositions). The lower right part of Figure 3.3 shows one such bushy decomposition. We can see that the corresponding candidate query plan is different and could not have been generated by the linear decomposition alone.

Our extend view matching algorithm can still leverage all of the optimizations proposed by Huang et al. [67], including considering co-partitioned edges as the smallest fragment since they can be computed locally, performing both linear and bushy decomposition of the queries, and applying cycle-based optimization.

3.4.2 View Selection via Query Planning

The assumption so far was that the user manually creates the views. The system then generates query plans to process pattern queries using one or more of them. However, in some cases, the user may not be able to select which views to create. The question therefore is whether the system can suggest users the views to create in such scenarios. Notice that the nice thing about recursive query planning is that we are anyways traversing the space of all possible query fragments that are relevant for the given query. We can consider each of these query fragments as candidate views. This means that every time we generate a query plan for a fragment, we add it to a global set of candidate views.

In the end, we can rank the candidate views using different utility functions and present the top ones. One such function could be the ratio of $cost$, i.e., savings due to the view, and $size$, i.e., the spending on the view. Recall that each query plan solution (the candidate view) contains its corresponding cost estimator as well, which could be used to compute these metrics. The system could also collect the candidate views over several queries before presenting the interesting ones to the user. We refer the readers to traditional view selection literature for more details [35].

The key thing to take away from here is that we can generate interesting candidate views as a side-effect of dynamic programming based query planning. This means that the user can start running his pattern queries on the input graph and later create one or more of the suggested views to improve the performance. To illustrate, we ran the view suggestion API for the six query workload from Figure 3.5. Table 3.1 shows the top-5 views of size three produced by the system.

3.4.3 Adaptive View Creation

We discussed how the system can help users to select views. However, the views are still created manually as an offline process. This is expensive and often the utility of a view is not known a-priori. Let us now see how we can leverage the query planning algorithm to adaptively create the graph views. The key idea is to start by materializing smaller query fragments and progressively combine them to create views of larger query fragments. To do this, we annotate each solution with the list of graph views it processes, i.e., solution s now have five pieces of information: $(\text{plan}, \text{cost}, \text{partitioning}, \text{estimator}, \{GV_i\})$. When combining the child query plans, we union the graph views from the children.

When the algorithm runs for the first time there is only a single input graph view which is the base graph itself. We look at all the leaf level query plans, and materializing the one(s) having the maximum utility, i.e., they are the most useful. In each subsequent runs, we consider materializing the query plans which combine existing views, i.e., we essentially *move* the view higher up in the query tree. We still consider materializing new leaf level plans from the base graph. Rather than combining the graph views greedily, a more fancy version can also keep counters on how many times each graph view is used. We can then combine the most frequent as well as most useful graph views.

The above adaptive view creation technique has two major advantages. First, it amortizes the cost of creating the view over several queries. This is because creating views at the lower levels involve fewer joins and hence it is cheaper. The system only spends more resources on creating a view in case it is used more often. Second, this approach starts from more general leaf level views, which could be used across a larger set of queries, and gradually specializes to larger views higher up in the query tree. This is useful in scenarios where a user starts from ad-hoc analysis and later converges to a specific query workload – something which is plausible in pattern matching queries.

3.5 Implementation

We implemented GraphFrames as a library on top of Spark SQL. The library consists of the GraphFrame interface described in Section 3.3, a pattern parser, and our view-based query planner. We also made improvements to Spark SQL's Catalyst optimizer to support GraphFrames.

Each GraphFrame is represented as two Spark DataFrames (a vertex DataFrame and a edge DataFrame), a collection of user-defined views. Implementations of each of the GraphFrame methods follow naturally from this representation, and the GraphFrame interface is 250 lines of Scala. The GraphFrame operations delegate to the pattern parser and the query planner.

Our query planner is implemented as a layer on top of Spark Catalyst, taking patterns as input, collecting statistics using Catalyst APIs, and emitting a Catalyst logical plan. At query time, the planner receives the user-specified views from the GraphFrame interface. The planner additionally can suggest views when requested by the user. The query planner also accepts custom attribute-based partitioners which it uses to make more accurate cost estimates and incorporates into the generated plans.

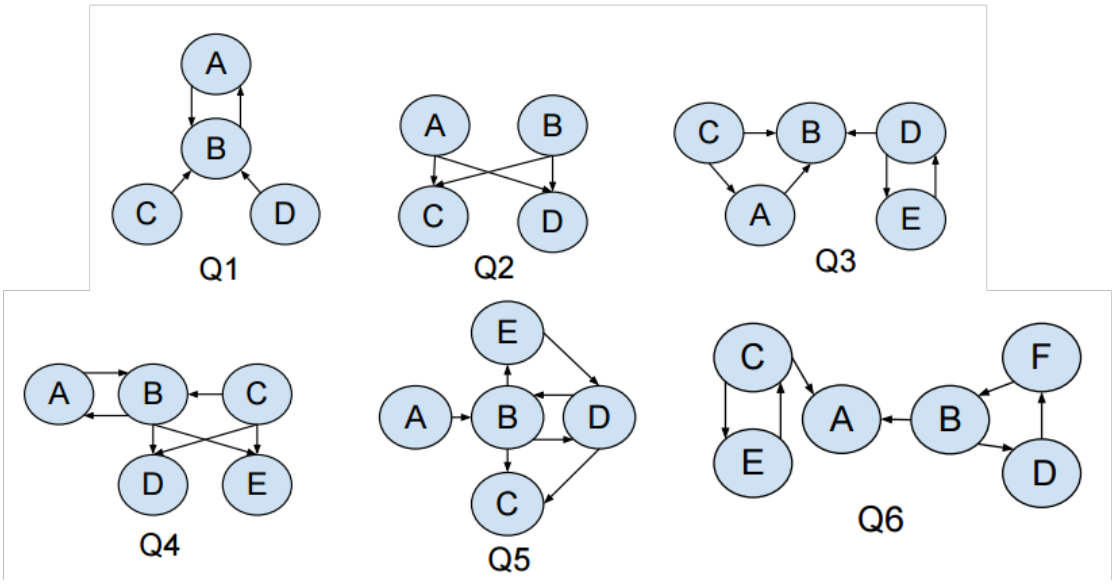


Figure 3.4: Pattern queries [67]

To simplify the query planner, we modified Catalyst to support join elimination when allowed by the foreign key relationship between vertices and edges. This change required adding support for unique and foreign key constraints on DataFrames to Spark SQL. Join elimination enables the query planner to emit one join per referenced vertex, and joins unnecessary to produce the final output will be eliminated. This change required 800 lines of Scala.

Our pattern parser uses the Scala parser combinator library and is implemented in 50 lines of Scala.

Finally, building on top of Spark enables GraphFrames to easily integrate with data sources and call its machine learning libraries.

3.6 Evaluation

In this section we demonstrate that GraphFrames match the performance of specialized graph query engines, greatly exceed their performance in some cases by materializing appropriate views, and outperform a mix of systems on analytics pipelines by avoiding communication between systems and optimizing across the entire pipeline.

All experiments were conducted on Amazon EC2 using 8 `r3.2xlarge` worker nodes in November 2015. Each node has 8 virtual cores, 61 GB of memory, and one 160 GB SSD.

3.6.1 Performance vs. Specialized Systems

We first show that GraphFrames match the performance of specialized graph query engines. We use the results reported by [67] as a baseline.

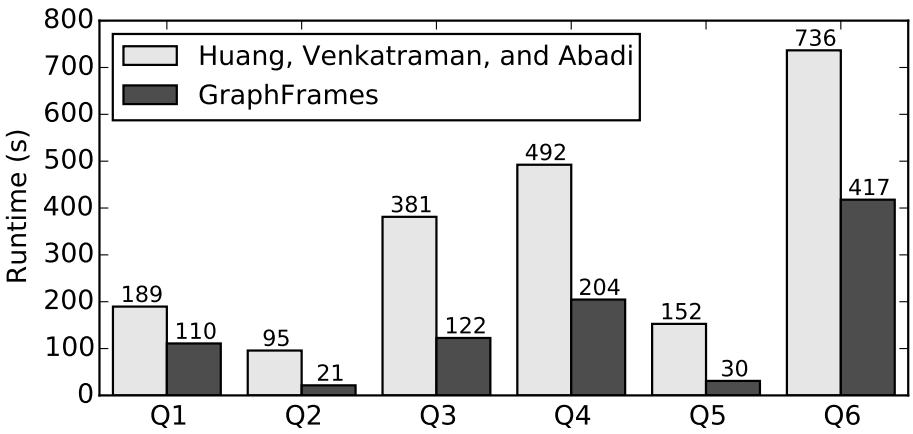


Figure 3.5: Performance of GraphFrames compared to a specialized engine for graph queries.

We ran the six queries shown in Figure 3.4 on a web graph dataset released by Google in 2002 [86]. This graph has 875,713 vertices and 5,105,039 edges. It is the largest graph used for these queries in [67]. In Figure 3.5 we compare the performance of GraphFrames to the results from [67], which were obtained on a 10-node cluster. GraphFrames match the baseline performance for each of the six queries. Since the two systems use identical query planning algorithms in the absence of views, differences in performance are due to the different cluster configurations and execution engines (PostgreSQL in the baseline compared to Spark in the case of GraphFrames).

3.6.2 Impact of Views

We next demonstrate that materializing the appropriate views reduces query time, and in some cases can greatly improve the plan selection. We reran the queries in Figure 3.4 on the same dataset. However, before running these queries we registered the views listed in Table 3.2. We then ran each query with and without view rewrite enabled. The results are reported in Figure 3.6.

Queries 1, 2, 3, and 6 do not benefit much from views, because the main cost in these queries comes from generating unavoidably large intermediate result sets. For example, in Query 1 the bidirectional edge between vertices A and B can use the 2-cycle view, but by far the more expensive part of the plan is joining C and D to the view, because this generates all pairs of such vertices.

However, in Query 4 we observe a large speedup when using views. In Query 4, the order-of-magnitude speedup is because the view equivalence check exposes an opportunity to reuse an intermediate result that the planner would otherwise miss. This is because the reuse requires recognizing that two subgraphs are isomorphic despite having different node labels, a problem that is difficult in general but becomes much easier with the right choice of view. In particular, the Triangle view is applicable both to the BCD triangle and the BCE triangle in Query 4, so the planner can replace the naive 5-way join with a single self-join of the Triangle view with equality constraints on vertices B and C.

View	Query	Size in Google graph
2-cycle	`(a)->(b)->(a)`	1,565,976
V	`(c)<-(a)->(b)`	67,833,471
Triangle	`(a)<-(b)->(c)->(a)`	28,198,954
3-cycle	`(a)->(b)->(c)->(a)`	11,669,313

Table 3.2: Views registered in the system to explore their impact on queries in Figure 3.4.

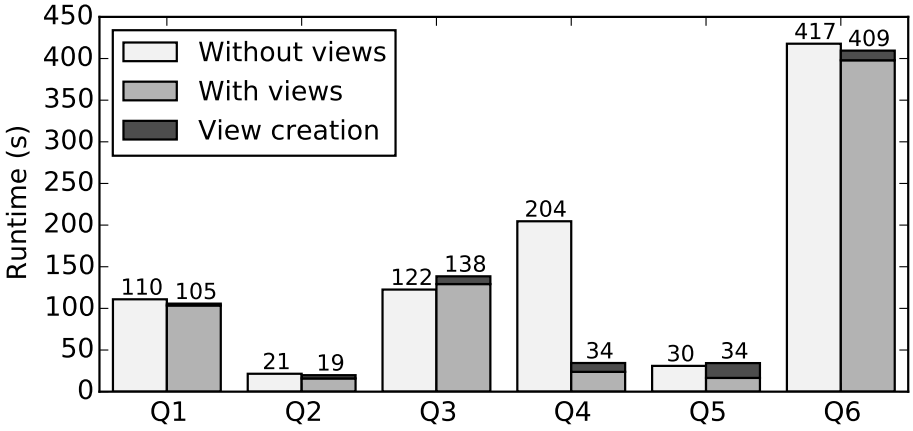


Figure 3.6: Performance of pattern queries with and without views

Additionally, in Query 5, precomputing views speeds up the main query by a factor of 2 by moving the work of computing the BCD and BED triangles from the main query into the Triangle 2 and 3-cycle views. These views are expensive to create, and since they are common patterns it is reasonable to precompute them.

3.6.3 End-to-End Pipeline Performance

We next evaluate the end-to-end performance of a multi-step pipeline that finds groups of users with the same interests in an Amazon review dataset. We will see that using Spark and GraphFrames for the whole pipeline allows more powerful optimizations and avoids the overhead of moving data between system boundaries.

We ran the pipeline described in Listing 3.3 on an Amazon review dataset [93] with 82,836,502 reviews and 168,954,245 pairs of related products. Additionally, after finding groups of users with the same interests in step 3, we aggregated the result for each user to find the number of users with the same interests. To simulate running this pipeline without GraphFrames as a comparison point, we ran each stage separately using Spark, saving and loading the working data between stages. In addition to the I/O overhead, this prevented projections from being pushed down into the data scan, increasing the ETL time. Figure 3.7 shows this comparison.

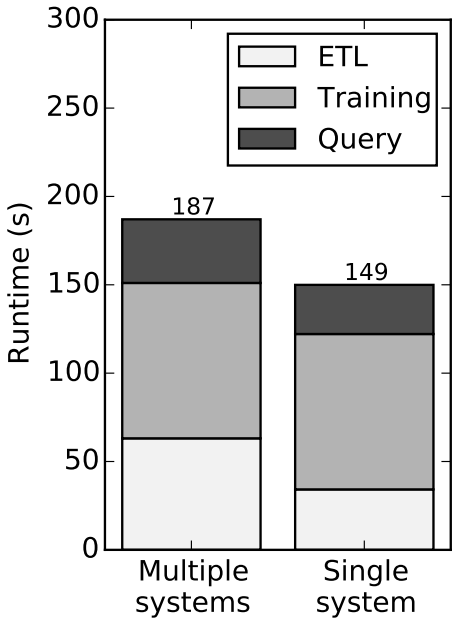


Figure 3.7: End-to-end pipeline performance in multiple systems vs. single system

3.6.4 Attribute-Based Partitioning Algorithm

GraphFrames supports partitioning based on arbitrary vertex or edge attributes, as discussed in Section 3.3.2. Its API enables users to optimize graph partitioning based on their domain-specific knowledges. We conduct an experiment to demonstrate the benefit of applying this API on the Amazon dataset [93].

We first apply random hash partitioning to the Amazon graph, and measure the average number of partitions each customer vertex belong to. This baseline represents the average number of times the system needs to replicate vertex data in GraphX, GraphLab, and GraphFrame. We then apply partitioning based on the product categories and measure the same metric, For items that belong in multiple categories, we only use the first category.

Figure 3.8 compares the average number of partitions for customer vertices in the two different partitioning schemes, using varying number of partitions. Partitioning by product categories leads to approximately two fold in data locality (i.e. half data communication).

3.7 Discussion

Although we have presented one implementation of GraphFrames, we believe that the abstraction is general and could benefit from other execution engines as well as additional optimizations. We discuss several ways in which the system can be extended.

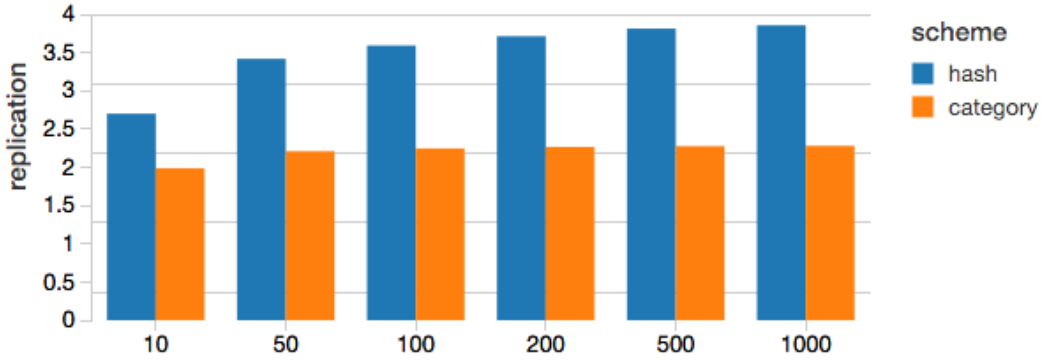


Figure 3.8: Benefit of Attribute-Based Partitioning

3.7.1 Execution on RDBMS Engines

Although we choose to integrate with Spark, GraphFrames could also execute over relational database engines, in a manner similar to Vertexica [75] and Grail [50]. All of the operations on GraphFrames compile into a relational operator tree, so they should be able to execute on an RDBMS engine. The main challenges in running GraphFrames over an RDBMS will be integrating our query planning algorithm (which may be possible through optimizer hints) and supporting user-defined functions (which we may replace with UDFs available in the database). Nonetheless, the system would retain the same high-level API for building workflows, and would benefit from more optimized performance and from the transactional, access control and recovery features of the DBMS.

3.7.2 Physical Storage

Since GraphFrames are built on top of Spark SQL, they are currently represented physically as vertex and edge tables using Spark SQL's compressed columnar representation [11] (and similar tables for any views). There are other physical representations that graph data may come in, such as sparse matrices, adjacency lists and triplets. We would like to explore using these as "first-class" formats for input data and letting the system work against these formats directly.

3.7.3 Additional Join Algorithms

Our current optimization algorithm produces a tree of pairwise join operators. As part of future work, we would like to support other options, such as one-shot join algorithms over multiple tables [1] and worse-case optimal join algorithms [36]. It should be possible to integrate these algorithms into our System-R based framework.

3.7.4 Dynamically Updating Graphs

GraphFrames currently do not provide support for processing dynamic graphs. In the future, we would like to develop efficient incremental graph update and processing support. We plan to leverage the newly available IndexedRDD [8] project to do this over Spark, or a relational database engine as an alternative backend. One interesting addition here will be deciding which graph views we wish to maintain incrementally as the graph changes.

3.8 Related work

To our knowledge, GraphFrames is the first system that lets users combine graph algorithms, pattern matching and relational queries in a single API, and optimizes computations across them. GraphFrames builds on previous work in graph analytics using relational databases, query optimization for pattern matching, and declarative APIs for data analytics.

Graph Databases Graph databases such as Neo4j [137] and Titan [132] focus on mostly on graph queries, often using pattern matching languages like Cypher [137] and Gremlin [119]. They have very limited support for graph algorithms such as PageRank and for connecting with relational data outside the graph database. GraphFrames use the pattern matching abstraction from these systems, but can also support other parts of the graph processing workflow, such as building the graph itself out of relational queries on multiple tables, and running analytics algorithms in addition to pattern matching. They then optimize query execution across this entire workflow. GraphFrames' language-integrated API also makes it easy to call user-defined functions (e.g., ETL code) and, in our Spark-based implementation, to call into Spark's built-in libraries, giving users a single environment in which to write end-to-end workflows.

Graph-Parallel Programming Standalone systems including Pregel, GraphLab, Trinity and X-Stream [91, 89, 124, 120] have been designed to run graph algorithms, but they require separate data export and import and thus make end-to-end workflows complex to build. GraphFrames use similar parallel execution plans to many of these systems (e.g., the Gather-Apply-Scatter pattern) while supporting broader workflows.

Graph Processing over RDBMS GraphX, Vertexica, Pregelix and Grail [59, 75, 25, 50] have explored running graph algorithms on relational databases or dataflow engines. Of these, GraphX materializes a triplets view of the graph to speed up the most common join in iterative graph algorithms, while the others use the raw tables in the underlying database. GraphFrames generalize the execution strategy in these systems by letting the user materialize multiple views of arbitrary patterns, which can greatly speed up common types of queries. GraphFrames also provide a much broader API, including pattern matching and relational queries, where these tasks can all be combined, whereas previous systems only focused on graph algorithms.

Query Planning for Pattern Matching Query planning for graph pattern matching has been studied in many domains, including SPARQL, social networks, and the semantic web; [125], [55] and [51] are some surveys on this work. Most of the early work assumes that the dataset fits in memory on a single machine, and does not consider distributed plans. Several recent projects have also explored distributed graph pattern matching. [128] and [143] optimize distributed pattern matching for large vertex-labeled graphs and Resource Description Framework (RDF) graphs, while [90] looks at graph simulation queries. Huang et al. [67] provide a System-R style dynamic programming algorithm for this task, which takes into account the characteristics of graph (e.g., that cycles will be more selective than arbitrary joins) to produce execution plans. Our work extends the algorithm in [67] to support multiple subgraph views as inputs to the planning process, and also proposes a mechanism for suggesting new views to build based on the user's queries.

Language-Integrated APIs Finally, the GraphFrames API is inspired by other declarative APIs for data processing, including Spark SQL DataFrames [11], Pig [110] and DryadLINQ [141]. These APIs are designed to be "programmer-friendly" by embedding into a procedural language or offering similar constructs, but still perform end-to-end relational optimization. Embedding into a procedural language makes it easier to build complex workflows in a modular function (e.g., by breaking them into functions) and to call UDFs. To our knowledge, GraphFrames are the first extension of this type of API to graphs, through our pattern matching, subgraphing and view creation operators.

Chapter 4

Arthur: Rich Post-Facto Debugging for Production Analytics Applications

Large-scale complex analytics increasingly features long-running queries, unreliable hardware and services, and heavy use of UDFs. This chapter proposes a debugger for distributed dataflow systems that addresses these challenges while incurring near-zero runtime overhead.

4.1 Introduction

Cluster computing frameworks such as Hadoop [6] and Dryad [72] have been widely adopted to enable sophisticated processing of large datasets. These systems provide a simple "data flow" programming model consisting of high-level transformations on distributed datasets, and hide the complexities of distribution and fault tolerance.

While these frameworks have been highly successful, debugging the parallel applications written in them is hard. To solve correctness or performance issues, users must understand the actions of thousands of parallel tasks, which produce terabytes of intermediate data across a cluster.

Debugging becomes especially difficult in *production* settings. Although tools for testing assertions, tracing through data flows, and replaying code exist (e.g., Inspector Gadget [109], Daphne [73], liblog [58], and Newt [43]), they invariably add overhead. For large-scale applications running 24/7, even 10% overhead can be expensive, so most operators do not use these tools in production, making bugs that occur in production time-consuming to diagnose and fix.

In this chapter, we present a new debugger, Arthur, that can provide these debugging features at close to zero runtime overhead, using *selective replay* of parts of the computation. Unlike previous replay debuggers for distributed systems [58, 63, 3, 4], which need to log a wide array of non-deterministic events (e.g., message interleavings) and are therefore expensive, we achieve our low overhead by taking advantage of the *structure* of modern data-parallel applications as graphs of deterministic tasks. Task determinism is implicitly assumed by the fault and straggler mitigation techniques in these frameworks [44, 72], but we use this same feature to efficiently replay parts of

the task graph for debugging.[1]

While the core idea of selective replay is simple, we show that it can be used to implement a rich set of debugging tools. These include:

- Forward and backward tracing of records through just the portion of the job that they affect.
- Interactive ad-hoc queries on intermediate datasets.
- Re-execution of any task in the job in a single-process profiler or debugger.
- Introduction of assertions or instrumentation (e.g., print statements) into the job.

To implement these features, Arthur must tackle several challenges. First, despite frameworks' assumption of task determinism, user error may cause *nondeterministic replay*, making it impossible to reconstruct a task's output. We do not aim to reproduce nondeterministic results, but instead detect them using checksums of task output across re-executions. We show that this checksumming adds minimal overhead to the original execution.

Second, to enable interactive debugging, Arthur also needs to be fast at replay time. We achieve high performance by (1) only replaying the subset of the job's task graph that is needed for a particular debugging action (e.g., to rebuild the input for one task), (2) parallelizing the replay over a cluster, and (3) caching frequently queried datasets in memory to provide fast access. As a result, many debugging queries can be answered within several seconds, even for large applications.

Third, Arthur's tracing feature requires *tracing records across a variety of operators* (e.g., map, filter, reduce, and group-by), taking into account the semantics of each operator. We perform tracing using a program transformation that augments each operator in the job to propagate tags with each record. To make the tracing efficient, we use a compressed tag set representation based on Bloom filters. A major advantage of our program transformation approach is that we do not need to modify the parallel runtime (Spark) to propagate tags.

We implement Arthur to support loading execution logs from either Hadoop or Spark [142] (a recent cluster computing framework with a concise Scala API). The system then replays the job in a Spark-based parallel runtime. It provides an interactive Scala shell from which the user can replay tasks, query intermediate datasets using arbitrary Spark queries, and run other analyses.

We evaluate Arthur on a variety of synthetic errors and three real bugs in Hadoop and Spark programs. The issues we test include logic bugs such as incorrect input handling, performance problems such as data skew, and unexpected program outputs that Arthur can trace back to specific input records. In all cases, Arthur's suite of tools can quickly narrow in on the problem. At recording time, Arthur adds less than 4% overhead and produces logs at most several megabytes in size. At replay time, Arthur finishes most analyses in a fraction of the running time of the original job, thanks to selective replay and in-memory caching, and supports querying intermediate datasets in sub-second time.

To summarize, our contributions are:

[1]Note that Arthur does not aim to debug non-deterministic problems, such as the user accidentally writing a non-deterministic task, but it will *detect* them using a checksum of each task's output. We show that this checksumming adds minimal overhead.

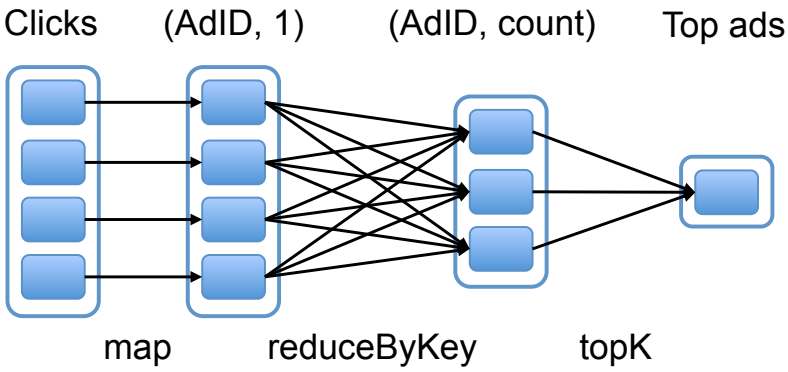


Figure 4.1: Dependency graph for the tasks produced in our example Spark program. Tall boxes represent datasets, while filled boxes represent partitions of a dataset, which are computed in parallel by different tasks.

- The observation that data flow frameworks' decomposition of jobs into deterministic tasks, which is fundamental in these systems for fault tolerance, can also be used to for low-cost replay debugging.
- A set of rich and efficient debugging tools that selectively replay only the part of the computation needed for an analysis, including ad-hoc queries and forward and backward record tracing.
- A fast, interactive environment for debugging jobs that allows ad-hoc queries in the Scala language and provides high responsiveness using parallel execution and in-memory caching.

The rest of the chapter is organized as follows. Section 4.2 describes the target environment for Arthur. We then discuss its architecture and capabilities in Sections 4.3–4.6. Section 4.7 discusses our implementation. We then evaluate Arthur (Section 4.8), discuss limitations and extensions (Section 4.9), and survey related work (Section 4.10).

4.2 Target Environment

Arthur is designed for parallel computing frameworks that use a data flow programming model, which include MapReduce [44], Dryad [72], Spark [142], Hyracks [21], Pig [110], FlumeJava [31], and others. It can also be applied to frameworks that are not typically thought of as data flow but do break computations into deterministic tasks, such as the Bulk Synchronous Parallel model in Pregel [91], where nodes operate on data locally and communicate at a barrier through the equivalent of a reduce operation. Many frameworks adopt this type of deterministic model for fault tolerance.

Arthur's replay approach takes advantage of the common assumption of task determinism, implicitly assumed for straggler mitigation and fault recovery, to provide accurate replay. Nevertheless, Arthur can detect nondeterminism and alert the user to the error, as described in Section 4.3.

To illustrate the structure of a data flow program, Figure 4.1 shows the lineage graph for a simple Spark program that computes the top 100 ads clicked in a log. This program's code is shown below:

```
val topAds = clicks.map(c => (c.adID, 1))
                   .reduceByKey((a, b) => a+b)
                   .topK(100)
```

This program uses Spark's functional API in the Scala language [142] to take a dataset called `clicks` (loaded, e.g., from a file) and run *map*, *reduce*, and *topK* transformations on it. The user code passed to these transformations (in this case, the Scala functions `c => (c.adID, 1)` and `(a, b) => a+b`) is expected to be deterministic.

Using Arthur, we can rerun just enough portions of the dependency graph to answer a particular debugging query. For example, if one of the *reduce* tasks is running slowly, we could replay all of the maps and that one reduce task, without having to replay the rest of the job.

4.3 Architecture

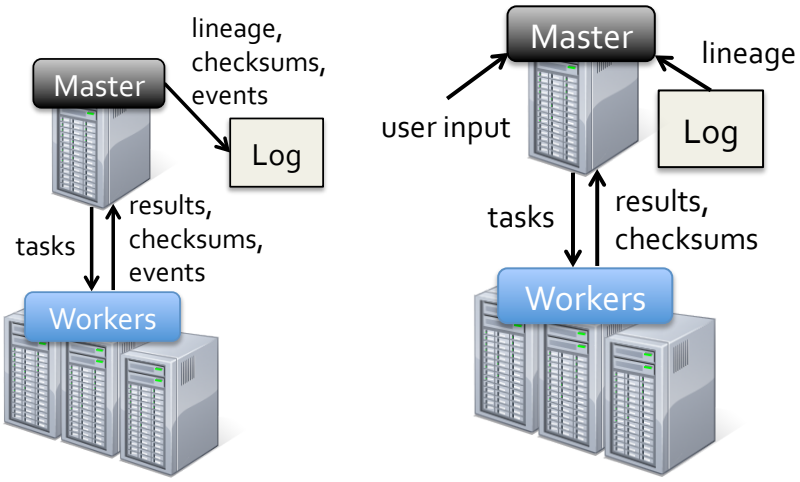
The main idea in Arthur is that we can *record* a data flow program's dependency graph at runtime, and *selectively replay* parts of the execution at debug time to answer users' questions. In this section, we describe how Arthur performs recording and replay.

At record time, Arthur runs as a daemon collocated with the cluster computing framework's master that logs several types of information. The most important is the program's *dependency graph*, which consists of every transformation in the program (e.g., the map and reduce in MapReduce or an operator in DryadLINQ or Spark) along with what input datasets or external files it acts on, and how it was partitioned into tasks. Arthur also records a *checksum of each task's output*; this allows the debugger to compare the checksums at replay time to the original ones, and alert the user that a task is nondeterministic if they differ. Finally, Arthur logs the execution time of each task, as well as the *cause of failure* (e.g., an uncaught exception) for any tasks that fail. Figure 4.2a summarizes the flow of information at record time.

After the program has finished running, the user launches Arthur in replay mode and loads the program's execution log. Arthur then accepts queries through an interactive shell and uses the recorded information to *replay* parts of the program's execution on demand. Replay takes place in parallel on the cluster; Arthur replays tasks from the appropriate parts of the dependency graph by launching them using Spark. Figure 4.2b illustrates the flow of information at replay time.

Arthur's basic replay functionality, described in Section 4.4, supports rerunning portions of the program exactly. This allows users to visualize the program's dependency graph, explore intermediate datasets, and rerun specific tasks locally in a conventional debugger.

Arthur also provides a more powerful type of replay that involves *modifying* the original operator graph. This makes it possible to perform analyses such as tracing records forward and backward through the data flow (described in Section 4.5). It also makes it possible to insert post-hoc instrumentation into the execution graph, such as assertions on intermediate datasets and other custom code (described in Section 4.6).



(a) Recording (b) Debugging

Figure 4.2: Flow of information while recording a program's execution and replaying and debugging the program. MapReduce, Dryad, and Spark carry out user transformations by deploying tasks onto the cluster and receiving their results. Arthur logs additional information about the program's execution, which it can replay on demand after the program finishes.

4.4 Basic Features

Arthur's core features are built on top of its ability to load the original program graph and replay parts of it on demand. The user accesses these features through an interactive shell based on Spark's Scala shell.

4.4.1 Data Flow Visualization

The simplest tool that Arthur provides is to use the lineage of datasets in the execution log to provide a visualization of the program's data flow graph. Such a visualization can be helpful in understanding the data access patterns and general structure of the program, and it only requires local analysis rather than re-execution of tasks in the job. Figure 4.3 shows an example lineage graph produced by Arthur on a PageRank application. Arrows point from datasets to their dependencies. The graph indicates that dataset 4 is used repeatedly in future computations, suggesting that it might be a good candidate to be cached in memory on the cluster, for example.

4.4.2 Exploration of Intermediate Datasets

In debuggers for programs on a single machine, variable inspector windows and "print" commands give visibility into a program's intermediate state. Arthur can provide a similar experience for data flow programs by allowing the user to query any intermediate dataset post-execution from the interactive debugger shell. To query an intermediate dataset, we (1) read the dependency graph that

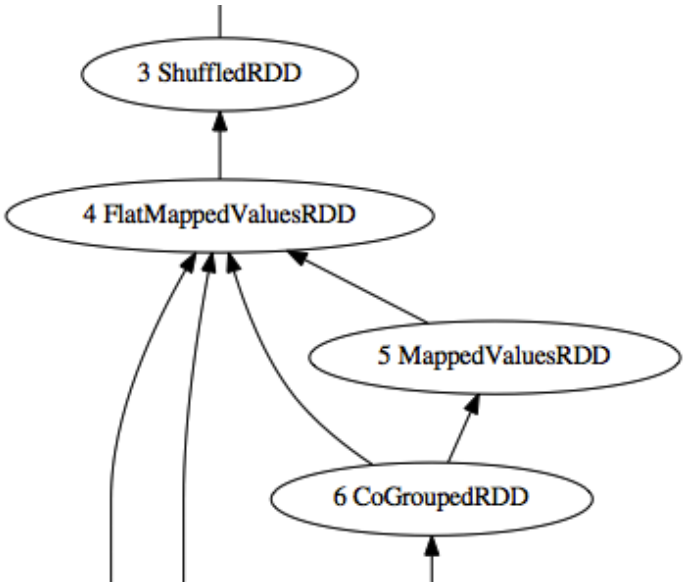


Figure 4.3: Partial lineage graph of a Spark application, as plotted by Arthur.

Arthur recorded, (2) find the tasks required to rebuild it, and (3) run them on workers across the cluster. Queries on datasets can be written using any of the operators in Spark [142], which include relational operators, sampling, and transformations using arbitrary Scala code.

For example, the following console session shows how a user might explore an intermediate dataset in the program from Section 4.2 that computes the top 100 ads clicked in a log. The user loads the `topads.log` execution trace and queries dataset 2, which is a collection of ads and the number of user clicks on each ad.

```
scala> val r = new EventLogReader("topads.log")
  r: EventLogReader = EventLogReader@726b37ad

scala> r.datasets(2).take(5) // sample first 5 ads
                             // and click counts
                             // from dataset 2
  res0: Array[(AdID, Int)] = [...]

scala> r.datasets(2).map(pair => pair._2).mean()
  res1: Int = 258288  // mean # of clicks per ad
```

Because Arthur executes its operations in parallel on the cluster, queries on intermediate datasets run at least as quickly as the original program, and frequently more quickly because only part of the job needs to run in order to produce the requested output. In addition, Arthur uses Spark's capability to cache information in memory once it is computed, enabling it to respond to repeated queries on the same dataset at interactive speeds.

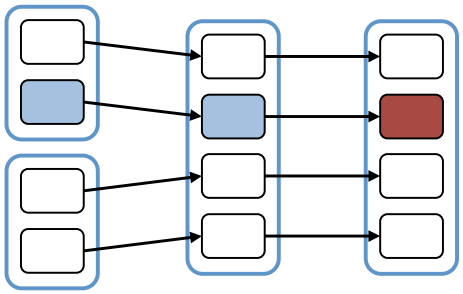


Figure 4.4: Tasks that need to be rerun for local task replay. To rerun a task (dark red), Arthur first runs its ancestors (light blue) and saves the last output. It is only necessary to run these tasks rather than the entire job.

4.4.3 Task Replay

Logic errors in bulk operators such as *map* functions can be difficult to debug because they execute in parallel on many machines. Programmers typically debug such operators by printing trace information from within the operator and later reading logs on machines that produced exceptions or incorrect results. Instead, it would be helpful to use conventional step-through debuggers and profilers. For example, if a certain task is throwing an exception on specific input, stepping through the user code in that transformation would make it easier to debug.

Arthur supports running specific tasks locally under such tools. To rerun a task locally, Arthur first computes the input to that task by running the tasks that it depends on; these tasks persist their outputs to disk. Arthur then launches a small wrapper program locally that receives the task metadata, fetches the outputs of parent tasks, and executes the task in an isolated environment. The user can then attach a conventional debugger such as JDB before the task runs, making it possible to set breakpoints, catch exceptions, and step through the operator's execution on the input data of interest.

Local task replay only requires a small portion of the program to be re-executed. In particular, only the task's ancestors must be run, rather than the entire program. Figure 4.4 shows that in order to debug an incorrect result from a particular task, Arthur only needs to rerun those tasks which contribute results to that task's input.

4.5 Record Tracing

Finding the set of records that stemmed from or led to a given record can be helpful in debugging the program's operations. For example, in a post to the Spark user group, a user described a word count application that unexpectedly output a count for the empty string in addition to the counts for each word in the input. Tracing that record backward through the program would reveal that the empty string stemmed from empty lines in the input, allowing the user to fix the input parsing bug. Arthur provides the ability to perform such tracing "post-hoc" by rerunning a *transformed* version of the original program.

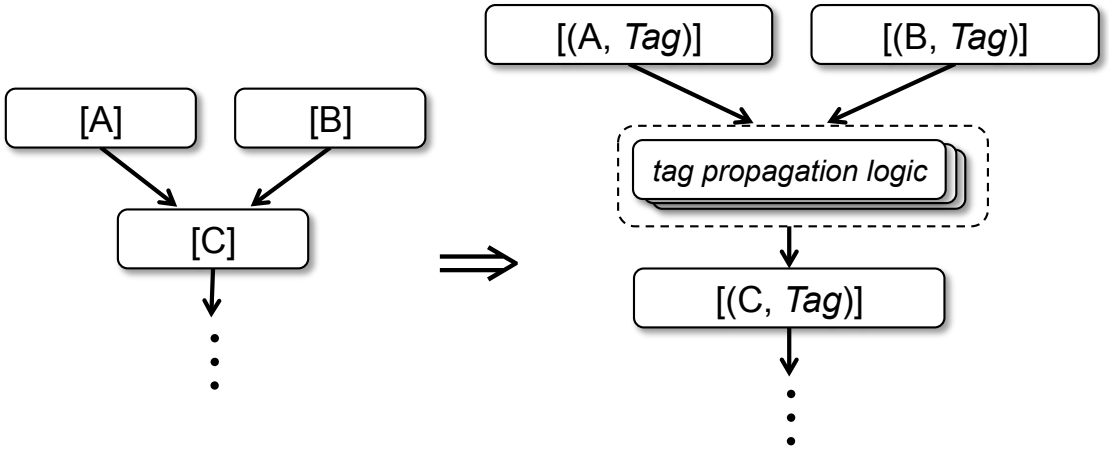
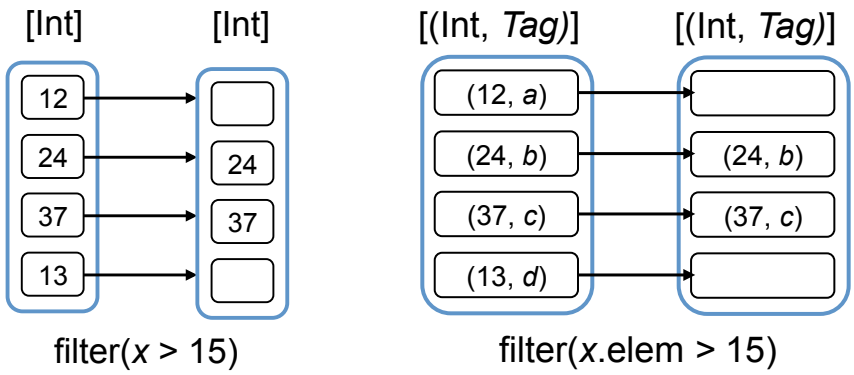


Figure 4.5: For tracing, Arthur rewrites the dependency graph to propagate *tags*, which represent provenance, along with each element. For example, the original dependency graph contains an operator that merges datasets of A and B elements to form a dataset of C elements. In the modified graph, the original operator is wrapped with logic to propagate the tags.



(a) *filter* on integers

(b) *filter* on tagged integers

Figure 4.6: The original *filter* operator applies a user-supplied predicate directly to each element, while the augmented operator extracts the integer element before passing it to the predicate.

4.5.1 Mechanism

Arthur provides record tracing using a program transformation in which it modifies each operation in the original program graph to propagate a *tag*, which represents provenance, along with each element. In addition to performing its previous function, each operator in the new execution graph is augmented to propagate tags, as illustrated in Figure 4.5. We implement forward and backward record tracing by using this tagging primitive to mark elements of interest and track their ancestors or descendants in the data flow, as described in the next sections.

The program transformation approach takes advantage of the functional, high-level nature of operations in modern data flow frameworks, which provide Arthur with precise information about how data moves through the program. To extract this information, it is necessary for the definition of each operator to include operator-specific tag propagation information. For example, Figure 4.6 shows that the *filter* operator propagates tags by extracting the element and passing it to the filtering function. In general, an operator f from datasets of type A to datasets of type B must also come with a function from the original operator f to an operator from datasets of type (A, Tag) to those of type (B, Tag).

This approach to record tracing relies upon the fine-grained semantics of dataset operations for accuracy. Operations like the *filter* above carry each input record to at most one output record, allowing the system to track records without any loss of fidelity. On the other hand, some frameworks, such as Hadoop, provide a coarser-grained API where a "map" function operates on multiple input records at once using an iterator, and writes output to a second iterator. Such operations expose only a coarse data flow structure, limiting the fidelity possible with a simple program transformation. More involved techniques such as static analysis of user-supplied operations could improve fidelity. In addition, for these types of operations, Newt [43] proposes a timing-based approach using the observation that any particular output record could only have been influenced by the input records that have appeared until that point. We use this approach in Arthur to handle Hadoop's map operator and a similar operator called *mapPartitions* in Spark.

4.5.2 Forward Tracing

Forward tracing is straightforward to implement on top of the tag-propagating program transformation. Arthur transforms the dependency graph into one that propagates a Boolean tag in addition to each record. It initializes the input records of interest with a `true` tag and other records with a `false` tag, runs the modified job using the Boolean *or* operation to combine tags, and finds which output records end up with a `true` tag. We show that forward tracing requires $< 1.5\times$ overhead at debug time, while leaving the original runtime unaffected.

When only a few records are of interest, Arthur traces them through just the relevant *subset* of the execution graph. For forward tracing, Arthur reruns the required tasks in each stage with tagging, inspects these tasks' outputs, looks up the elements' shuffle keys to determine which tasks in the next stage read records from these tasks, and repeats the process on the new set of tasks.

4.5.3 Backward Tracing

Like forward tracing, backward tracing builds upon tag propagation. Because operators are not guaranteed to be invertible, backward tracing cannot simply tag output records with booleans and run the program in reverse. Instead, it tags each input element with a *unique* tag, runs the job, examines the tags that ended up on the output records of interest, and finds which input elements contributed those tags. This process is illustrated in Figure 4.7.

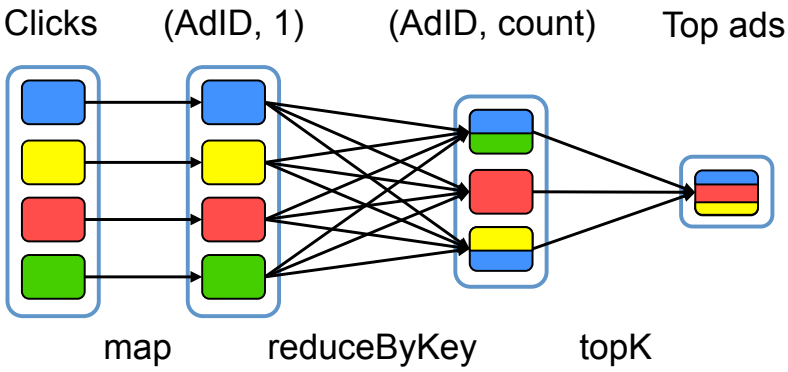


Figure 4.7: To trace an output record (rightmost rectangle) *backward* through the data flow, we tag each input element uniquely, run the job to propagate the tags to the outputs, and find which input elements contributed tags (leftmost blue, yellow, and red rectangles).

We implement unique tags using integers. Each input records is tagged with a unique integer based on its position in the dataset, and tags are stored as sets of integers which are combined using the *union* operation.

This approach works well for programs such as database queries where each record has a clear provenance. However, in iterative programs such as PageRank, each output record is influenced by a large number of input records. As a result, tags tend to diffuse widely, and in the extreme case each output record may end up with a tag from every input record. This approach therefore performs poorly for long jobs because of the high space overheads that tags impose in later stages. Representing tags as Bloom filters provides a 1.78× speedup on PageRank, at the cost of false positives.

An alternative approach to backward tracing is to trace the output records of interest backward through each stage, starting with the last stage. In each stage, Arthur tags each record in the shuffle output from the previous stage with a unique label, runs the stage, and finds which input elements contributed to the tags of the output elements under consideration. This approach allows the tag dependency structure to be precomputed, allowing further backward tracing queries to be performed using just a single cluster-wide join operation.

4.6 Post-Hoc Instrumentation

Arthur's ability to replay a *modified* version of the data flow graph, which we used in tracing, also opens the door to other types of analyses. One that we have explored is *post-hoc instrumentation* of the code, where assertions or print statements can be added to part of the job after it was executed and can be verified in the debugger. While step-through debugging and tracing are powerful tools for tracking down problems, often the best way to understand program execution, especially in a large application, is to test assertions, and Arthur provides the ability to inject these without runtime overhead.

The simplest type of assertion one can test is about the contents of a particular dataset. For example, suppose that we were debugging a PageRank application, and we wanted to ensure that the PageRank of each node was positive at all iterations. We could attach an assertion to the dataset for a given operation as follows:

```
scala> val ranks = r.datasets(2) // dataset ID #2

scala> ranks.assert(r => r > 0)
```

Arthur will test the assertion by adding a no-op *map* operator after the computation of `ranks` that verifies the predicate and reports any records that do not pass it to the master. By default, Arthur's assertions are attached "lazily" (they are not tested right away), so it is possible to attach multiple assertions to multiple datasets, and then test all of them using a `EventLogReader.checkAssertions()` function. Because Arthur's shell is simply a Scala interpreter, it is also possible to attach these assertions programmatically (e.g., search through the list of datasets for all the ones created on a particular line of the program and add assertions to all of them).

Apart from these types of data assertions, Arthur also allows users to instrument their code more closely by replaying a *modified* version of the original binary. As long as the functions in the program still produce the same outputs (i.e., any modifications are only for print statements or assertions), the system can still run the program on the cluster. Currently, this modification has to be done manually before starting Arthur, but we also wish to support dynamic modification of the program code using the Java VM's class reloading feature [87] in the future.

Finally, it would be straightforward to extend the assertion mechanism to support "distributed assertions" (in the form of a Spark expression that has to hold for an entire dataset, e.g., that the sum of PageRanks is close to 1) [57]. Currently, these can be checked using manual Spark queries on the datasets, as in Section 4.4.2.

4.7 Implementation

We implemented Arthur in about 2000 lines of Scala code. The system supports recording applications written in either Hadoop or Spark, and replaying them in a Spark-based parallel runtime where different debug operations can be invoked interactively from a Scala shell.

At recording time, Arthur needs to obtain (1) the graph of operators used in the parallel job and (2) checksums of intermediate datasets, used to verify that re-execution has been deterministic. In Hadoop, the operator graph is trivial, because it is always a single MapReduce step, so we only use the job's configuration to rerun the same map and reduce functions. In Spark, we added an event logging module that logs the datasets used in each parallel operation to a file. For checksums, we use a simple Java OutputStream wrapper that computes a checksum as data is written out. We only perform checksumming for data at task boundaries (e.g., for the output data in a map task), where it is either written to a file or sent over the network, so the checksumming adds little overhead because the cost of sending the data over the network is much more expensive.

At replay time, Arthur replays both Hadoop and Spark computations in the Spark engine, to take advantage of features such as in-memory caching and interactive queries. We use an existing layer on top of Spark, called SHadoop, to execute Hadoop map and reduce tasks within Spark. (This is conceptually simple because Spark also supports map and reduce operators.) We begin by loading the job's code from a path provided by the user, followed by an event log with the parallel operations run during the job and their operator graphs (as discussed above), then present the user with an interactive Scala shell where they can view the operations and datasets and run queries on them. The actual replay of both the original operators and any transformed versions of them (e.g., for assertions or tracing) is implemented by submitting jobs to the existing Spark engine, so it does not require changes to Spark.

The debugging interface for Arthur is a shell in the Scala programming language, based on Spark's interactive Scala shell. It provides an object model to load and debug jobs, and lets the user define local variables or functions using the complete Scala language, and query datasets using functional operators written in Spark. Users can also explicitly control which datasets to cache in memory for repeated queries. For example, the following console session shows how one might query an intermediate dataset in a PageRank computation, whose log is read from `pagerank.log`, and replay a task:

```
scala> val r = new EventLogReader("pagerank.log")
  r: EventLogReader = EventLogReader@726b37ad

scala> r.datasets
#00: hadoopFile at PageRank$.main(PR.scala:31)
#01: map        at PageRank$.main(PR.scala:31)
#02: map        at PageRank$.main(PR.scala:35)
#03: groupBy    at PageRank$.main(PR.scala:35)
#04: flatMap    at PageRank$.main(PR.scala:35)
#05: map        at PageRank$.iterate(PR.scala:91)
#06: cogroup    at PageRank$.iterate(PR.scala:92)
[...]

scala> r.datasets(2).count()
  res0: Long = 129941   // # of elements in dataset 2

scala> r.debugTask(3, 1) // replay task 1 of dataset 3
[launches the JDB debugger]
```

Finally, for replaying individual tasks in a single-process debugger, we wrote a small wrapper program that reads a serialized Task object from Spark and reads its input from a file (computed in parallel using the cluster), then executes that task locally, so that we can spawn this program in a local subprocess and attach JDB or other debugging tools to it.

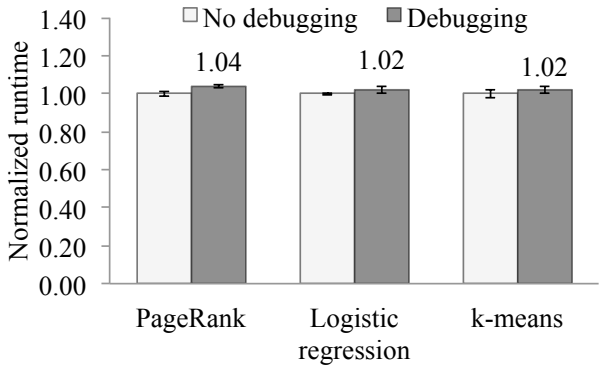


Figure 4.8: Performance comparison of various Spark applications with and without debugging.

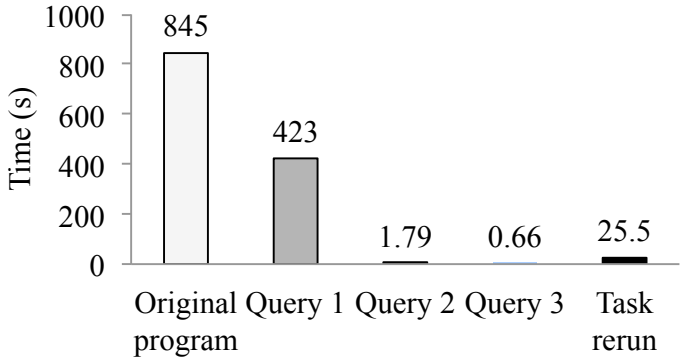


Figure 4.9: Running time of various interactive queries in the debugger. Arthur only runs the tasks necessary to answer each query, so Query 1 is faster than the original program. Subsequent operations benefit from in-memory caching.

4.8 Evaluation

We evaluated Arthur using a variety of real bugs and injected errors in Hadoop and Spark programs. We find that Arthur's overhead at record time is only 2–4%, making it feasible to run continuously. At debug time, we found that Arthur can respond to queries at interactive speeds (on the order of a second) by caching frequently-used datasets in memory, and often needs to rerun only a subset of the job even for the first time it answers a query. Finally, we evaluated Arthur's applicability to different types of bugs by showing how it can diagnose task failures, incorrect output, nondeterministic behavior, and deterministic performance problems in various programs.

4.8.1 Recording Overhead

We tested Arthur's recording overhead on three Spark programs: PageRank, logistic regression, and k-means. Our PageRank program runs 15 iterations on the articles in a 54 GB Wikipedia dataset, our logistic regression program runs 10 iterations on 10 million 10-dimensional vectors of

doubles, and our k-means program runs 5 iterations to cluster 100 million 4D points into 4 clusters. Figure 4.8 compares the runtimes of these applications with and without Arthur when running on a 20-node cluster. Because the overhead from task output checksumming constitutes most of Arthur's recording overhead, applications with large task outputs see higher overhead than those with small task outputs. Each PageRank iteration updates the rank for every page, forcing Arthur to checksum a large amount of information. On the other hand, each iteration of logistic regression only outputs a single vector, and each iteration of k-means only updates a small number of cluster centers. Regardless of application, however, Arthur's logging overhead was at most 4%, and its log size less than 5 MB.

4.8.2 Replay Performance

At replay time, Arthur can often run faster than the original program thanks to selective replay and in-memory caching. Figure 4.9 shows the running times of various queries when debugging the PageRank program. Query 1 counts the number of articles whose PageRank in iteration 3 is greater than a threshold. Arthur only needs to run the first three iterations to answer the query, so it runs faster than the original program. Next, Query 2 calculates the average PageRank in the third iteration, and Query 3 sample the PageRank of ten articles. These queries reuse the same dataset as Query 1, so they benefit from in-memory caching. Finally, rerunning a single task locally was fast compared to the original program.

We also benchmarked the performance of forward and backward record tracing during replay of the PageRank application. Forward tracing was 1.48× slower than the original program, while backward tracing with integer set tags was 5.52× slower. The large slowdown for backward tracing was due to diffusion of tags, as described in Section 4.5.3. Both versions correctly identified the records that an input depended on (e.g., in tracing through three iterations of PageRank, we find neighbours at most three links away). We also tested backward tracing with Bloom filter tags. This reduced the slowdown to 3.09× the original program due to the more compact size of the Bloom filter representation, but caused about 30% of records in the output to be erroneously tagged. The actual performance ratio and error rate depends on the size of Bloom filter used.

4.8.3 Applicability

Though Arthur is focused towards debugging deterministic problems, we have observed these to be more common than nondeterministic errors for complex distributed programs due to the fact that, like MapReduce and Dryad, Spark requires transformations to be deterministic. To illustrate the kinds of errors that Arthur can detect, we describe its applicability to three deterministic errors: task failures, incorrect output, and performance problems. We also describe Arthur's ability to detect unintended nondeterminism through checksumming on a real bug from Conviva, as well as its support for loading Hadoop traces on a pre-existing bug in Mahout.

Deterministic Task Failures As an example of a deterministic task failure, we injected an input processing bug into our Wikipedia PageRank program. This program extracts the links from each

article by parsing its XML representation and searching the parse tree for link elements. Certain tasks were consistently throwing XML parsing exceptions, so we used Arthur to search the event log for failing tasks. We reran one of the tasks in a conventional debugger and found that the culprit records contained `\N` in place of the article's XML. It turned out that our Wikipedia dump used this string to represent an empty article, so we fixed the bug by adding error-handling code for that case.

Incorrect Output As an example of incorrect output, Carat, a real Spark program for processing time series of devices' power usage, contained a bug causing the output to contain nonsensical negative values for power usage. We used Arthur to inject assertions at every stage and then replayed the program. Arthur detected the assertion failures immediately after the location of the bug and we were able to halt the program early, avoiding the inconvenience of rerunning the program in its entirety.

Deterministic Performance Problems We debugged Monarch [131], a real Spark program that used logistic regression to classify spam status updates. The program was running much more slowly than expected, and we observed that a few straggler tasks were consistently finishing last, tens of seconds later than the typical task. We recorded the task IDs that appeared to be stragglers and, once the program had finished, we loaded its trace into Arthur and reran the tasks locally under JDB. Examining the input partition to the task revealed that it contained several very large feature records, identifying partition skew as the source of the problem. We were able to reproduce the problem because Spark shards data into partitions deterministically.

Unintended Nondeterminism We used Arthur to detect a bug arising from unintended nondeterminism at Conviva, a video analytics company. An analytics query intended to operate on new records that had arrived in the last few minutes was performing the filter by comparing record timestamps against the current system time from within the query, as in the following example:

```
records.filter((System.currentTime() - _.time)
               < INTERVAL)
```

When we used Arthur to replay the query, we received checksum mismatch warnings because the time had changed from the original execution, and the query now matched fewer records. Examining the operator that triggered the warnings revealed the bug, and we fixed the bug by computing the time in the driver program instead of in the operator, so that the reference time would remain the same across executions of the task:

```
val now = System.currentTime()
records.filter((now - _.time) < INTERVAL)
```

Hadoop Jobs To demonstrate Arthur's support for loading Hadoop job traces, we chose a preexisting bug in Mahout [7], a Hadoop-based machine learning library. We reproduced the bug, MAHOUT-363, in Arthur. This bug involved a NullPointerException due to a logic error in the *map* code within Mahout. We were able to load the affected Hadoop job into Arthur, identify the map

task causing the error, and rerun and step through the task locally until the NullPointerException. Inspecting the task code and state in JDB confirmed that the exception was due to the Mahout Cache's failure to handle a null feature vector.

4.9 Discussion

Arthur can perform detailed analysis of job executions with nearly zero runtime overhead by leveraging the determinism and structure of modern data-parallel applications. While the core idea behind Arthur is simple, we showed that it efficiently supports a wide range of analyses, which can be sped up by only replaying the relevant parts of the job. We believe that Arthur's approach is important for two reasons. First, because the deterministic data flow model we exploit was primarily adopted for fault tolerance, we believe that it will remain present not only in today's frameworks (e.g., MapReduce, Dryad, Spark), but in future ones as well. Indeed, it is interesting that not only the determinism itself, but also the decomposition of jobs into small tasks, is used to *speed up recovery on failure* (by minimizing the work redone), and both elements directly speed up selective replay. Second, because of the intrinsically high hardware cost of big data computations, any instrumentation at runtime is expensive, so replay may be the *only* effective way to debug production problems. Therefore, it is important to study which analyses can be performed this way.

Because of its reliance on replay, Arthur does have limitations that more invasive debuggers would not. We discuss some of these next, followed by ways in which Arthur can be extended. We also discuss how parallel runtimes could be extended to enable easier replay.

4.9.1 Limitations

Arthur's replay approach has several limitations, some of which can be avoided with more care during execution:

Nondeterministic User Code Arthur cannot replay bugs where a user's code (e.g., a map function) is nondeterministic, although it *detects* them using checksumming. While users of data flow frameworks are asked to try to write deterministic code to enable fault recovery, nondeterminism can still be a bug, so it is important to be able to fix it. There are two interesting possible approaches. One is that, once nondeterminism has been detected, Arthur can try to *reproduce* nondeterministic behavior (though maybe not the same as the original run) by simply running multiple copies of the problematic task. It could also run these tasks in a more expensive replay debugger, such as R2 [63], that can recreate nondeterministic events once it seems them the first time. A second approach would be to try to identify nondeterministic code through static analysis (e.g., see whether particular libraries are being called).[2]

[2]The most common nondeterministic library that might be called is a random number generator, but fortunately, runtimes *can* make that deterministic by seeding the generator consistently for a given task ID. For example, Spark does this for its built-in *sample* operation.

Inter-Task Interactions Sometimes, bugs are not caused by a particular task, but by the interaction between multiple tasks on the same machine. For example, Hadoop runs a series of tasks in the same Java VM to amortize startup costs, but if each task leaks memory or uses a library with global state, the behavior of a task may depend on which others have run before it. Arthur cannot guarantee to run the same tasks together at replay time (especially if doing selective replay). Auxiliary monitoring tools, such as a memory usage monitor, might be used to detect some of these conditions.

Lost Input Files Arthur implicitly assumes that the input files for each job are still available at replay time. Fortunately, most data warehouses operate in an "append-only" fashion, and retain files for a long time after ingestion. HDFS does not even support random updates.

Communication Order A subtle issue that can happen in some frameworks is that even though the code in each task is deterministic, an instance of that task might fetch input data from other tasks in a nondeterministic order. For example, in Spark, a reduce task performing a commutative operation (such as a sum) fetches results from multiple map tasks in parallel and receives chunks from different tasks in different orders. Although the operation is, technically, expected to be commutative, some bugs might manifest only depending on the input order. In our implementation, we modified Spark to log the order of chunks fetched and use the same order at replay time. For Hadoop, this problem is not present because Hadoop always sorts the input to a reduce function.

Nondeterministic Programming Models While Arthur works for many current frameworks that perform deterministic computations, such as MapReduce, Dryad, Spark, Hyracks, and Pregel, it cannot be applied to programming models that allow nondeterministic, asynchronous messaging, like MPI or Graphlab [60].

4.9.2 Extensions

While we have implemented several useful debugging tools in Arthur, we discuss other analyses that would be interesting to implement in the model, especially by taking advantage of the parallelism of the cluster, in Section 5.2. In addition, if replay is going to be the cheapest way to debug production problems, it would also be interesting to extend runtime frameworks to better support it.

4.10 Related Work

Debuggers for Data Flow Frameworks Two recent systems for debugging parallel data flow programs are Inspector Gadget [109] and Daphne [73].[3]

Inspector Gadget is a debugger for programs in the Pig scripting language that adds instrumentation into the program to monitor various properties (e.g., the time spent in each task, the number of records matching a predicate, or user-specified assertions). However, this approach requires the

[3] "Arthur" was chosen to continue this trend of cartoon characters.

Property	Inspector Gadget	Daphne	liblog/R2/ODR	Arthur
Job visualization	✓	✓	✗	✓
Queries on intermediate data	✗	✗	✗	✓
Local task replay	✗	✓*	✓	✓
Assertions	✓	✗	✓	✓
Profiling	✓	✓	✓	✓
Record tracing	✓	✗	✗	✓
Runtime overhead	5–70%†	minimal	> 20%	< 5%

Table 4.1: Comparison of Inspector Gadget, Daphne, general replay debuggers, and Arthur. Note that (*) Daphne's task replay requires that all intermediate data in the job is saved to disk and available at debug time, and (†) Inspector Gadget requires instrumenting jobs at runtime, with varying overhead based on the analysis done.

user to instrument their job before they run it, and does not allow the user to rerun a task in a local debugger or to run ad-hoc queries on intermediate datasets that she did not add instrumentation for in advance. The runtime overhead from instrumentation can be as high as 70% for some analyses (e.g., data sampling and latency analysis using tags), making it expensive to run in production. In contrast, our replay approach lets users ask ad-hoc questions about the job *after* it finished, including rerunning parts of the job with the same kinds of instrumentation available in Inspector Gadget, and additionally supports local step-through debugging of tasks.

Daphne lets users visualize and debug DryadLINQ programs. Daphne provides a "job object model" for viewing the tasks in a job, hooks for attaching a debugger to a remote process on the cluster, and the ability to replay a task in a single-process debugger *as long as its input data is still available on the cluster*. This approach works in DryadLINQ because all communication between tasks is through files on disk, but it will not work in the increasing number of frameworks that perform computations in memory (such as Pregel [91], Graphlab [60], or Spark [142]), or for jobs where the intermediate data has been deleted. In contrast, Arthur can *recompute* the input to any task. In addition, Arthur also provides checksumming to verify that the user's code runs deterministically (an assumption in Daphne) and a rich set of capabilities that are not present in Daphne because they require running new code on the cluster, such as running ad-hoc queries on intermediate datasets.

In general, our implementation provides a superset of the features in these debuggers, and shows that these features can be be implemented "post-facto" using selective replay of the parts of the job that a particular feature requires. Other features in Arthur, such as the ability to reconstruct intermediate datasets and run ad-hoc queries on them, or to add new assertions after the job has finished, are unique in our approach because they require running new computations on the job's intermediate data without knowing these computations during the original job's execution. Table 4.1 summarizes the features in Arthur in comparison to other debuggers.

Record Tracing and Provenance Several projects have explored how to efficiently capture provenance of records in data-parallel computations to enable tracing. RAMP [70] defines and captures provenance for "generalized map and reduce workflows," which are programs composed of an acyclic graph of map and reduce steps. However, because it does this tracing during job execution, it can add substantial runtime overheads (20–76%). Newt [43] is a provenance capture and replay framework for Hadoop and Hyracks that supports capturing record-level provenance at runtime, and then replaying just the part of the job that produced a particular output record. Unlike RAMP, it also handles map operators that work on a stream, by looking at the interleaving of records being read and written to determine which input records affected an output record. (RAMP assumes that the map function processes just one record at a time and cannot maintain state between records.) However, Newt still incurs about 14–26% runtime overhead, and it lacks other debugging functions, such as checking assertions or running ad-hoc queries on intermediate datasets. Inspector Gadget [109] supports forward tracing through tag propagation and backward tracing by tagging all input records, but it again needs to perform this tagging at runtime.

All of these approaches could be used within Arthur to capture provenance for records in (part of) the job when recomputing it, while avoiding the runtime overhead in production. Our current tracing module uses the properties of Spark and Hadoop operators, as well as a Newt-like approach for map functions that operate on iterators.

Replay Debuggers Replay debugging for distributed systems has been extensively studied through systems such as `liblog` [58], R2 [63], ODR [3], and DCR [4]. However, these systems are designed to replay *general* distributed programs, and thus work by recording all sources of nondeterminism, including message passing order across nodes, system calls, and accesses to memory shared across threads. This results in significant overhead at runtime (often more than 20%), or even larger slowdowns at replay time ($> 10\times$) for systems that log fewer events but infer the order of missing events [3]. In contrast, our debugger leverages the *structure* of datacenter computing frameworks to deterministically replay tasks. This approach allows us to catch a large class of logic and performance bugs, and although we cannot replay some of the nondeterministic bugs that other systems capture (e.g., race conditions between threads in the same task), we can still detect them via checksumming. Our recording overhead is also low enough that event logging can be turned on by default in production.

Chapter 5

Conclusion

5.1 Highlights

5.1.1 Oblivious Coopetitive Queries (OCQ)

We proposed OCQ, an efficient, general framework for oblivious coopetitive analytics using hardware enclaves. OCQ's contributions are its query planner design, which supports flexible party-specific sensitivity rules; its mechanism for propagating and refining padding upper bounds based on foreign key constraints; and its mixed-sensitivity join algorithm. Due to its performance gain over existing solutions, we believe OCQ is the first realistically practical system for secure coopetitive analytics.

5.1.2 GraphFrames

Graph analytics applications typically require relational processing, pattern matching and iterative graph algorithms. However, these applications previously had to be implemented in multiple systems, adding both overhead and complexity. We aimed to unify the three with the GraphFrames abstraction. The GraphFrames API which is concise and declarative, based on the "data frame" concept in R, and enables easy expression and mixing of these three paradigms. GraphFrames optimize the entire computation using graph-aware join optimization and view selection algorithm that generalizes the execution strategies in previous graph-on-RDBMS systems. GraphFrames are implemented over Spark SQL, enabling parallel execution on Spark and easy integration with Spark's external data sources, built-in libraries, and custom ETL code. We showed that GraphFrames make it easy to write complete graph processing pipelines and enable optimizations across them that are not possible in current systems.

GraphFrames was released as open source in 2016 and has garnered significant interest on GitHub, with 594 stars at the time of writing.

5.1.3 Arthur

As cluster programming frameworks are adopted for more applications, debugging the programs written in them is increasingly important. This is challenging both because of the scale of the applications and because of the cost of the hardware resources involved, which makes any runtime overhead for debug information expensive. We have proposed an approach based on *selective replay* that exploits the deterministic nature of computations in these frameworks to efficiently rerun parts of the program. We show that this approach enables a rich set of analyses, including rerunning tasks in a conventional step-through debugger, checking assertions, tracing records forward and back through the computation, and interactively querying intermediate results. The cost to log the operations we require is minimal (less than 4%), allowing our recording to be "always on" in production use. The deterministic operations we leverage are a crucial element of current programming frameworks because they enable fault recovery [44], so we believe that they will remain present in future frameworks, making our approach applicable there as well.

5.2 Future Work

Secure vectorized query execution. Vectorized query execution [20] is a natural fit for secure hardware enclaves because it (1) avoids the need to compile and attest new code at query time, (2) makes heavy use of branchless operations that reduce side-channel leakage, and (3) achieves high performance with a simple system design that reduces the trusted computing base compared to code generation.

Oblivious user-defined functions. OCQ provides oblivious implementations of relational operators, but does not support UDFs. It would be useful to attempt to enforce obliviousness for UDFs or to transform UDFs into oblivious form using techniques from programming languages. The goal is to avoid the overhead of MPC frameworks by verifying that a UDF is inherently oblivious.

Support for hybrid queries spanning trusted hardware and cryptography. Certain threat models may call for both trusted hardware and cryptography. For example, trusted hardware can be used to enforce integrity with low overhead, while cryptography can be used when confidentiality is needed. A planner that could determine when to apply each of these techniques could greatly reduce overhead compared to a crypto-only solution.

Vectorized worst-case-optimal join algorithms for graph queries. Worst-case-optimal join algorithms avoid the need to materialize large intermediate result sets during graph querying. However, they are traditionally expressed in a tuple-at-a-time model, which is not ideal for maximizing memory parallelism. A vectorized worst-case-optimal join algorithm could be ideally suited for high-performance graph querying.

Adaptive planning for graph queries. Graph data typically exhibits extreme skew, making it a potential fit for adaptive query processing techniques [46]. Batching input data offers an opportunity to amortize the cost of runtime plan changes across many tuples. This could allow graph queries to coexist in the same system as traditional SQL workloads rather than resorting to specialized systems and join algorithms for graphs.

Parallel Profiling Arthur provides a very effective foundation to run shadow profiling [101], a technique where multiple copies of the program are run in parallel with sample profiling to collect highly detailed statistics. Arthur's model naturally allows doing this for only a subset of the job (e.g., one task) and feeding the same input to every copy.

Minimal Example Discovery When a deterministic bug, such a task crashing, occurs, it would be useful to automatically "narrow down" on a smaller example that produces the problem by trying smaller subsets of the job's input. This approach is taken in some existing testing tools, such as QuickCheck [37], and clearly benefits from a parallel search.

Extending Runtimes for Debuggability Some of the limitations we highlight in Section 4.9.1 lead directly to ways to extend parallel runtimes for easier debuggability (and, ultimately, more chance of recovering correctly from faults as well). Some ways that frameworks could help Arthur include isolating tasks from each other in separate processes,[1] providing hooks to fetch task inputs in a specific order (as we have done in Spark), and retaining intermediate data on the filesystem after job completion (if the framework normally writes temporary files and then deletes them), and allowing this to be used as an input during replay to avoid recomputation. Most of these changes would make programs in these frameworks easier to understand in general.

[1] One reason this was not done in Hadoop is due to the startup cost of new Java VMs, but this does not need to be a fundamental limitation.

Bibliography

[1] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[2] R. Agrawal and R. Srikant. Privacy-preserving data mining. *SIGMOD Rec.*, 2000.

[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[4] G. Altekar and I. Stoica. DCR: Replay-debugging for the datacenter. Technical Report UCB/EECS-2010-33, UC Berkeley, 2010.

[5] Apache drill. `https://drill.apache.org/`.

[6] Apache Hadoop. `http://hadoop.apache.org`.

[7] Apache Mahout. `http://mahout.apache.org`.

[8] Apache Spark. Spark IndexedRDD: An efficient updatable key-value store for apache spark. `https://github.com/amplab/spark-indexedrdd`, 2015.

[9] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, Jan. 2013.

[10] ARM. Trustzone. `https://developer.arm.com/ip-products/security-ip/trustzone`.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[12] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[13] B. Bahmani, A. Chowdhury, and A. Goel. Graph data models for genomics. *IEEE Engineering in Medicine and Biology special issue on Managing Data for the Human Genome Project*, 1995.

[14] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 2013.

[15] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure querying for federated databases. *Proc. VLDB Endow.*, 2017.

[16] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers. Shrinkwrap: Efficient sql query processing in differentially private data federations. *Proc. VLDB Endow.*, 12(3), 2018.

[17] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[18] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[19] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[20] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[21] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE '11*, pages 1151–1162, 2011.

[22] T. Bourgeat, I. A. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *IEEE/ACM Internationl Symposium on Microarchicture*, 2019.

[23] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[24] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[25] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 8(2):161–172, Oct. 2014.

[26] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[27] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX SECURITY*, 2017.

[28] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, Microsoft Research, June 2010.

[29] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[30] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.cms.hhs.gov/hipaa/, 1996.

[31] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI '10*. ACM, 2010.

[32] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 2017.

[33] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019.

[34] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[35] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *The VLDB Journal*, 11(3):216–237, 2002.

[36] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.

[37] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP '00*, 2000.

[38] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[39] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. Usenix Security Symposium, 2016. https://eprint.iacr.org/2015/564.

[40] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[41] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–8, 2016.

[42] A. Dave, C. Leung, R. A. Popa, J. E. Gonzalez, and I. Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[43] S. De, D. Logothetis, and K. Yocum. Scalable lineage capture for debugging DISC analytics. OSDI poster session, 2012.

[44] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[45] H. S. Delugach and T. H. Hinke. Wizard: A database inference analysis and detection system. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[46] A. Deshpande, Z. Ives, V. Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[47] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, Mar. 1990.

[48] E. Eizenman. Scotiabank's chief risk officer on the state of anti-money laundering. In *McKinsey Company*, 2019.

[49] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[50] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[51] W. Fan. Graph pattern matching revised for social network analysis. In *ICDT*, 2012.

[52] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. Don't reveal my intension: Protecting user privacy using declarative preferences during distributed query processing. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, 2011.

[53] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. PAQO: Preference-aware query optimization for decentralized database systems. In *2014 IEEE 30th International Conference on Data Engineering*, 2014.

[54] D. Froelicher, P. Egger, J. S. Sousa, J. L. Raisaro, Z. Huang, C. V. Mouchet, B. Ford, and J.-P. Hubaux. Unlynx: A decentralized system for privacy-conscious data sharing. *Proceedings on Privacy Enhancing Technologies*, 4:152–170, 2017.

[55] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.

[56] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-preserving distributed linear regression on high-dimensional data. *PoPETs*, 2017(4):345–364, 2017.

[57] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *NSDI '07*, pages 21–21, 2007.

[58] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX ATC*, 2006.

[59] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, and I. Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[60] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI '12*, 2012.

[61] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[62] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[63] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI*, 2008.

[64] R. Hall, S. E. Fienberg, and Y. Nardi. Secure multiple linear regression based on homomorphic encryption, 2011.

[65] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *SIGMOD*, 2008.

[66] T. H. Hinke. Inference aggregation detection in database management systems. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*, 1988.

[67] J. Huang, K. Venkatraman, and D. J. Abadi. Query optimization of distributed pattern matching. In *ICDE*, 2014.

[68] T. Hunt, Z. Jia, V. Miller, C. J. Rossbach, and E. Witchel. Isolation and beyond: Challenges for system security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, 2019.

[69] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[70] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR 2011*, 2011.

[71] M. Ion, B. Kreuter, E. Nergiz, S. Patel, S. Saxena, K. Seth, D. Shanahan, and M. Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. `https://eprint.iacr.org/2017/738`.

[72] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[73] V. Jagannath, Z. Yin, and M. Budiu. Monitoring and debugging DryadLINQ applications with Daphne. In *HIPS*, 2011.

[74] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[75] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: Your relational friend for graph analytics! *Proc. VLDB Endow.*, 7(13):1669–1672, Aug. 2014.

[76] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for SQL queries. *Proc. VLDB Endow.*, 2018.

[77] N. M. Johnson, J. P. Near, J. M. Hellerstein, and D. Song. Chorus: Differential privacy via query rewriting. *CoRR*, abs/1809.07750, 2018.

[78] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, Apr. 2016.

[79] A. F. Karr, X. Lin, A. P. Sanil, and J. P. Reiter. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics*, 2005.

[80] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[81] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus, 2019.

[82] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems (Eurosys)*, 2020. `https://keystone-enclave.org/`.

[83] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[84] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX SECURITY*, 2017.

[85] T. Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 71–80. ACM, 1984.

[86] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[87] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA '98*, pages 36–44, 1998.

[88] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[89] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In P. Grünwald and P. Spirtes, editors, *UAI*, pages 340–349. AUAI Press, 2010.

[90] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, 2012.

[91] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[92] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[93] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 785–794, New York, NY, USA, 2015. ACM.

[94] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[95] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[96] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[97] Microsoft. Always Encrypted database engine, 2019. `https://msdn.microsoft.com/en-us/library/mt163865.aspx`.

[98] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, 2018.

[99] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019.

[100] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[101] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07*, pages 198–208, 2007.

[102] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[103] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

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

[105] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[106] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in MapReduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[107] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[108] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[109] C. Olston and B. Reed. Inspector Gadget: a framework for custom monitoring and debugging of distributed dataflows. In *SIGMOD*, 2011.

[110] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[111] `pandas` Python data analysis library. http://pandas.pydata.org.

[112] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[113] B. Parno, J. Lorch, J. J. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[114] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[115] Presto. `https://prestodb.io/`.

[116] X. Qian, M. E. Stickel, P. D. Karp, T. F. Lunt, and T. D. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 196–205. IEEE, 1993.

[117] The R project for statistical computing. http://www.r-project.org.

[118] J. L. Raisaro, J. Troncoso-Pastoriza, M. Misbach, J. S. Sousa, S. Pradervand, E. Missiaglia, O. Michielin, B. Ford, and J.-P. Hubaux. Medco: Enabling secure and privacy-preserving exploration of distributed clinical and genomic data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2019.

[119] M. A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

[120] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[121] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15. IEEE Computer Society, 2015.

[122] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[123] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[124] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[125] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 39–52, New York, NY, USA, 2002. ACM.

[126] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[127] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Data Engineering, 1994. Proceedings. 10th International Conference*, 1994.

[128] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[129] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[130] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[131] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[132] Titan distributed graph database. `http://thinkaurelius.github.io/titan/`.

[133] K. U. verification to disentangle secure-enclave hardware from software. Andrew ferraiuolo and andrew baumann and chris hawblitzel and bryan parno. In *SOSP*, 2017.

[134] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[135] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[136] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. Cryptology ePrint Archive, Report 2017/189, 2017. `https://eprint.iacr.org/2017/189`.

[137] J. Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA, 2012. ACM.

[138] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*. Springer, 2016.

[139] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEESP*, 2015.

[140] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 2017.

[141] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[142] 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 *NSDI*, 2012.

[143] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276. VLDB Endowment, 2013.

[144] M. Zhao, P. Liu, and J. Lobo. Towards collaborative query planning in multi-party database networks. In P. Samarati, editor, *Data and Applications Security and Privacy XXIX*, 2015.

[145] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[146] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[147] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1), Aug. 2009.