Literature Review: Making Multiparty Computation Industry-Viable for Data Processing and Retrieval

Nick Young
Brown University
nicholas_young@brown.edu

1. OVERVIEW

Multi-party computation (MPC) allows multiple parties to compute over shared data without compromising the privacy or integrity of any party's data. Many industries benefit from being able to derive insights from a wider pool of data without learning more than they need to; common examples include hospitals jointly computing biomedical statistics without leaking confidential patient information; or financial firms jointly computing market share data without leaking information about their firm's behaviour. In particular, parties can jointly answer SQL queries over shared data using techniques including garbled circuits [12], oblivious RAM [6], or secret sharing [2]. While we could generate protocol-specific MPC for each query type we may want to answer, a more generic and flexible solution is to develop techniques to optimize our use of MPC in query execution.

Work has been done to scale MPC to a higher number of parties; however, scaling to a higher number of inputs to make this computation feasible for industry applications is an area of active research. Interestingly, one of the approaches that helps scale MPC to large workloads is to find ways to minimize its usage. Given that MPC is much slower (on the order of $10^5\times$ to $10^7\times$ slower [19]) than local computation, the more systems can operate on data in the clear or offload computation to a trusted party, the less unnecessary compute they will incur. In this literature review, we've identifies three main categories of techniques that help reduce MPC usage in SQL query evaluation.

First, we have techniques with roots in database query optimization, informed by work in information flow control. By annotating data with fine-grained privacy controls, we can be very selective with how much data is revealed to other parties, potentially leading to huge speedups by allowing more computation to be done outside of MPC through rewriting query plans. Traditional query optimizations don't necessary work in the MPC setting, and so work has been done to redefine a set of MPC query optimizations. These optimizations range from rule-based to cost-based.

Second, we have optimizations in query execution itself. Enabled by knowledge on which sets of data should be available to which parties, a suite of optimizations that allow partial or parallelized execution of queries in the MPC setting have blossomed. With minimal sacrifice in the security properties of our system, we can execute many operations much faster by offloading work to trusted parties.

Third, we have technologies inspired and enabled by the development of secure enclaves. A recent surge in secure hardware that allows isolated and verifiable execution at processor speeds enabled the development of new oblivious execution primitives for query execution. These primitives allow us to use MPC with more aggressive security models, leveraging the fact that the hardware itself is secure.

In this literature review, we review the techniques and technologies used to improve performance of MPC in data retrieval and processing and compare them to alternatives for secure data processing. In particular, we focus on query optimization, information flow control, query execution, and oblivious secure enclaves.

2. BACKGROUND

Many industry players are in the game of big data processing, in which incredibly large datasets are crunched to derive business insights or make decisions on the fly. In the trusted setting, data processing is primarily done using distributed query evaluation engines such as Spark [17] over datasets stored in a distributed relational database. These engines typically expose a SQL front-end to answer queries and abstract away the rest of the details. The benefit of using a distributed query execution engine is to enable much faster query response times even as datasets grow arbitrarily large by scaling horizontally. Many industry players have grown around these engines and rely on them heavily for their data processing needs.

However, these engines all assume a trusted setting, where all of the data is owned by the same party. A common example for *multi-party* data processing is medicine [14] [3]. Often, hospitals collect the same data as their peers; data about patient morbidity, race, medical history, scans, etc. However, patient protection laws including HIPAA prevent hospitals from sharing raw data with their peers. This severely limits their capacity to derive insight from their

joint data, especially for research. It's clear that MPC would be useful in this setting.

Let's attach some formal terms to concretize our setting, which we borrow from [3]. A **federated database** is a database in which shards or tables are owned by different parties that may not be mutually trusting. A **shard** is a subset of the data in a **table**, which inherits its usual definition. To exercise these definitions, consider two scenarios:

1. In the hospital scenario, each hospital holds a shard of each table, but no hospital controls any one table in full. The federated units are the individual hospitals.
2. In the financial scenario, one federated unit might be a firm under audit, while the other might be the FCC. Each unit may control different tables in full, or a shard of some table.

2.1 Software environment

We wish to use MPC in an ergonomic fashion to answer database queries in the same way that engines like Spark do. Using a standardized interface such as SQL with an appropriate backend, we can abstract away the underlying multiparty computation so that parties can interact with data as if it were totally 'in the clear'. On top of providing a more ergonomic interface, a standardized query language enables more advanced and optimized query plans to be built over MPC primitives, creating ample room for query processing and optimization to improve the feasibility of using MPC in industry. The techniques we explore will assume a common SQL-based front-end and will largely apply optimizations to the query plan to speed up computation.

For completeness, it's worth describing some of the MPC frameworks and techniques commonly used and referenced in the literature. **Garbled circuits** allow two or more parties to evaluate an arbitrary circuit in an oblivious manner; no party will learn more than the output, and what can be inferred from their inputs and the output [12]. Because garbled circuits allow evaluation of an arbitrary circuit, they allow evaluation of arbitrary functions, which make them an excellent primitive for flexible query execution. In practice, a framework called **Obliv-C** is used to compile an extension of C into garbled circuits [18]. The framework is easily extendible and can be used to implement a variety of primitives including **ORAM** [6].

Secret shares allow parties to split up secret values into private pieces that can be computed on locally or with a little bit of communication, while maintaining secrecy [2]. After the secret shares have been computed on, they can be recombined to recover the final value. In practice, a framework called **Sharemind** is used to implement secret sharing [4]. Another framework called **Wysteria** combines both of these techniques in a domain-specific language [13]. All of these frameworks are well cited and used in the literature.

The primary barrier to using MPC in query processing is that MPC is very slow; on the order of $10^5\times$ to $10^7\times$ slower than computing in cleartext [19]. Query processing workloads are already prone to taking a long time to complete; the slowdown incurred by naively applying MPC makes this

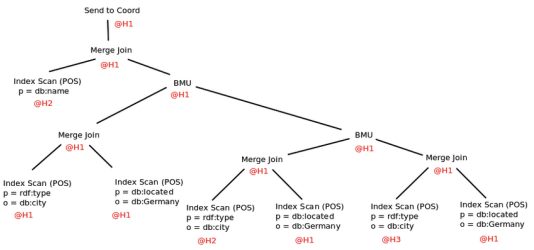

Figure 1: Example query plan

an infeasible option. Thus, we explore ways to apply MPC in an informed manner.

2.2 Security model

Most of the literature assumes a semi-honest static adversary, somewhat understandably. For big-data processing, there may be a number of parties wanting to jointly compute, but in general they will all be known and, therefore, able to be held accountable. For processing of data of many unknown parties, other technologies like fully homomorphic encryption or blockchains may be a better fit. However, some systems claim a straightforward path to security in the malicious setting [14], and we will see shortly that we can build primitives that work in stronger settings using specialized hardware [19].

The attacks that we are principally concerned with guarding against are information leakage and access pattern attacks. Protecting against information leakage, as we will see, comes down to being very careful that the optimizations we apply are safe. The definition of what information is okay to leak may also change depending on the framework we inspect, and thus we take care in labelling which data is truly sensitive. We will typically eschew security proofs in favor of those cited in the literature. Access pattern attacks boil down to an adversary inferring features of our data using memory or network access patterns. Such attacks are protected against using oblivious access patterns.

3. QUERY OPTIMIZATION AND INFORMATION FLOW CONTROL

3.1 What is query optimization?

Query optimization is the practice of taking a query plan and applying rewrite rules to speed up response times without affecting the output. A query plan is typically represented as an AST with leaf nodes as data tables and nonleaf nodes as relational operators such as joins or selects; see Figure 3.1 for an example of a query plan. Parsing a common front-end like SQL into an AST is a well-solved issue [16].

Query optimization is well studied. The classic optimization example in the trusted setting is predicate pushdown, wherein filters are pushed through joins to be evaluted first. By evaluating cheap filters before expensive joins, the database system has fewer entries to process in later steps. Moreover, notice that this doesn't change the rows in the result, so long as the predicate is only being evaluated on attributes from one of the input tables to the join [11].

Another classic optimization example in the trusted setting is join reordering. If we know the relative sizes of tables apriori, we can choose to evaluate joins that will output fewer rows first. This will lead to smaller relations being inputted into later joins [11].

Optimizations can be split into two categories; **rule-based** and **cost-based**. A rule-based optimization is one that is applied to a particular pattern present in a query plan; predicate pushdown is one such example. These optimizations are valuable because they are typically evergreen and provably correct. A cost-based optimization, on the other hand, leverages metadata about the relative sizes of data tables and calculates heuristics to compare and choose between query plans; join reordering is one such example. These techniques are being applied to both database query plans and MPC execution plans in state-of-the-art data processing systems [1].

Many traditional query optimization techniques apply to the MPC setting as well, in slightly modified forms. One could apply traditional statistics and heuristics-based cost optimization to decide between query plans. However, query plan cost must be evaluated with MPC in mind; clearly, query optimization with MPC involved requires insight into the runtime and security properties of MPC to best optimize the query while only revealing as much information as is safe to do. For example, join reordering may not be evergreen in the MPC setting, as it may cause early evaluation of sensitive data, forcing more work to be done in MPC as opposed to in the clear. In order to evaluate our query plans with MPC in mind, we need to give it more information. We do so using trust annotations and information flow control.

3.2 What is information flow control?

Information flow control (IFC) is the practice of tracking the lifetime of software objects through an execution and maintaining invariants on those objects. Work has been done to use IFC techniques to preserve application security by protecting data as it flows through the program [15]. Critically, many data flow assertions can be made at compile time. Obliv-C uses similar techniques to disallow compilations that would lead in leaking private variables (marked `obliv` in the Obliv-C syntax). Many systems have a notion of tagging data with information about its privacy level and enforcing that privacy level across the program. Research has been done to enforce policies through a policy engine or through the type system itself [10].

In the query evaluation setting, we can apply IFC by tagging input relations with their owners. From there, we can infer ownership over intermediate variables to better understand the set of parties that should be privy to certain information. In more advanced systems, not only can we tag tables with owners, but we can specify the privacy levels of specific columns as well as either public or private. Moreover, we can also specify a set of *selectively-trusted-parties* (STPs) for columns [14]. With these tags, each column will end up with a *trust set* of one or more parties that are authorized to view the data in the clear; all other parties shouldn't learn anything about the data. The idea behind such fine-grained trust annotations is that we can exercise even more liberty with optimizations the more we know about our query.

Using these trust annotations on input relations, we can derive the owner of each intermediate relation in a query plan. The owner of a child relation should be the intersection of the owners of the parent relations. Consider some cases; if a child relation is derived from a single parent relation, the owner of the parent would have been able to run the operation themselves. So, they are already privy to all information in the child relation, making them a reasonable owner. If a child relation is derived from two parent relations, the child relation would leak information about both input relations; thus, only those that owned both parent relations should own the child relation. This is called *forward propagation* [14].

We can also derive the owner of a parent relation given its child relation if the operation is *reversible*; that is, the parent is fully derivable from the child. An example reversible operation is scalar operations by known constants. This is called *backwards propagation* [14]. Such techniques allow us to fully infer the ownership and trust sets of all intermediate relations in a query plan; in future steps, we assume that this inference has already been run.

Some systems also propagate trust by inferring ownership of input relations using foreign-key relationships [19]. To ensure that we don't inadvertently leak correlated information through a foreign-key relationship, the system performs *second-path analysis*, which ensures that any further trusted tables don't leak information by verifying that any foreign-key relationships point *into* the sensitive data columns, and not out of. Thus, the trusted tables can't implicitly embed sensitive information.

3.3 Rewriting execution plans

Before we dive into how query optimization and IFC can apply in the big data query setting, we first take a look at how these techniques have informed MPC itself. MPC compilers take a program in some language (either domain-specific in the case of FairPlay or Wysteria, or an extended language in the case of Obliv-C) and compile it into an execution plan, that may then be turned into garbled circuits. However, the program itself can often be rewritten in ways to get the compiler to create a faster program. The techniques specified in [9] model these programs as a simplified AST, then apply rewrite rules exploiting associativity or distributivity to improve MPC performance.

A key insight is that in an MPC program, each variable belongs to some subset of the parties. Suppose we apply the IFC tagging and inference techniques as laid out above. Notice that any variable that belongs to all parties (i.e. has all labels) can be computed on in the clear, and any variables that are being combined with only variables that share the same owner can be computed locally. Moreover, notice that there isn't an obvious way to compute on values in the clear in the middle of a query plan without leaking information; in other words, we can expect our query plans to have a single 'core' of MPC, prepended and appended by in-the-clear computation (this assumption isn't actually true if we relax our security definition, but suffices for now). Thus, if we can rewrite our execution plans to have a maximal amount of computation at the beginning and the end, more work can be done in the clear and less in MPC. Note that

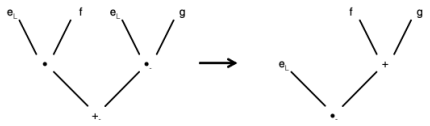

Figure 2: Forward application of distributive law

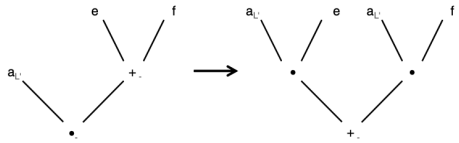

Figure 3: Backward application of distributive law

not all of our rewrite rules will actually reduce the total amount of work done; but, it will try to move the work into settings where it can run quicker.

We now explore some optimizations for a simplified arithmetic language. Assume we have a language with variables and binary operators which may be associative, commutative, and distributive over each other (e.g. addition and multiplication). One clear optimization we can make leveraging associativity and commutativity is to flatten all operations of the same type and ensure that all inputs with a common owner are combined first. That is; assume we have $2n$ input variables a_i, b_i that are all being combined in a sum: $\sum_i (a_i + b_i)$. We would much rather evaluate this sum as $(\sum_i a_i) + (\sum_i b_i)$, since we then incur one MPC-enabled task instead of n.

The distributive law also allows us to eliminate excess MPC-enabled operations. As shown in Figure 3.3 and Figure 3.3, we can distribute either forward to eliminate an operation or backward to create more local computation with a shared public value. Both reduce the amount of MPC-enabled computation in favor of local computation.

It's clear that these optimizations should speed up our execution plans, but how do we choose between them? This problem is complicated by the fact that the space of optimizations is not smooth; some optimizations may enable or disable others. A heuristics-based approach, similar to that of real database query optimization, works well. By estimating the cost of each execution plan and choosing the minimum, we can arrive close to the optimal manually-optimized execution. We can estimate the cost of each operator using a table like in Figure 3.3, where λ is the security parameter, or length of inputs [9]. The details of estimating cost are complicated and implementation specific, so we eschew a lengthy discussion on them.

3.4 Rewriting query plans

We now turn our attention to rewriting query plans in the MPC setting. Many of the insights we've generated thus far still apply. The main optimizations we are going to make will reduce the amount of MPC being used by maximizing the work done in cleartext in the beginning and end of the query plan.

Operator	Cost
Labeled	0
Unlabeled	
Comparison	λ
Addition†	λ
Multiplication†	λ^2
Exponentiation	λ^3

Figure 4: Cost table

Let's first consider a simple example of join reordering. By riffing on the original optimization, we can use join reordering to first combine multiple public tables, then combine the result with a secret table; this can reduce the number of secure joins from 2 to 1. Since joins commute, this is a perfectly correct optimization to make [19].

Using rules like this and others like predicate pushdown, moving the point at which MPC must be used towards the center of the query plan is known as *MPC frontier pushdown* or *push-up* [14].

4. QUERY EXECUTION

There are a whole host of optimizations that we can apply to relational operators themselves to make them run faster. Indeed, especially when armed with knowledge about which parties can be trusted with which data, we can offload data in the execution phase to a trusted party that can do work for us in the clear. We explore a suite of optimizations that take advantage of what we've learned from query optimization and IFC.

4.1 Splitting and slicing

Notice that filters and aggregates are **splittable** in the federated database setting. That is, we can split an arbitrary filter or aggregate into partial evaluations that can first be partially evaluated at a particular node then finished in MPC, minimizing the amount of work done using MPC. We detail how we can split filters and aggregates.

We could naively compute filters over a shared table by sending all of the data into MPC and then running filters for each tuple over all attributes. Alternatively, we could partially evaluate a filter on all of the insensitive attributes locally first, then use MPC to evaluate the filter over sensitive attributes. To see why this is secure, notice that evaluating the public attributes doesn't contradict the security model, since public attributes can't leak data about private attributes (if they did, we must have tagged our attributes improperly to incur a functional dependency). Thus, whether we do it in MPC or not doesn't matter, so pulling it out is valid.

Similarly for aggregates, we could compute partial aggregates over each input shard, then compute the full aggre-

gate from the partial aggregates using MPC. Notice that this only works for associative aggregate functions (e.g. min, max, and count, but not average). To see why this is secure, recall that no party ever shares sensitive data with another party. Thus, they will also not be able to reconstruct group-by data from the other party. While parties may be able to infer statistics about other parties using the output, this is acceptable under the security model.

Notice also that joins don't need to be evaluated using MPC unless the join key is present in more that one data provider. So, we compute all joins locally within a partition, so long as that partition owns all of the rows of the join key. Since no other party has inputs to this operation to begin with, the security of this optimization boils down to nobody learning the output; since the output will either be used as an MPC input (and therefore hidden) or revealed, this property holds.

Lastly, we can cleverly *slice* our input data to allow parallelizing query execution. Each operator decides on a *slice key* which should be an expression over public attributes that can divide up computation evenly. Each *slice partition* is evaluated separately; care is taken to ensure that a slice key doesn't change the overall result of the operation.

4.2 Hybrid operators

Given information about each columns selectively-trusted parties (STPs), we can try to split work-intensive operations like joins and aggregations into *hybrid operators*, which outsource some portions of execution to a STP. We define and flesh out two hybrid operators: *hybrid joins* and *hybrid aggregation* [14].

Hybrid joins can be used if the key columns of both sides of a join share a STP. The steps are as follows:

1. Parties obliviously shuffle the input relations.
2. Parties send only the key column to the STP.
3. STP enumerates the key-column relations it receives, assigning an ID to each row.
4. STP computes a cleartext join on the enumerated key-column relations.
5. STP secret-shares the result of the cleartext join to untrusted parties.
6. Parties perform oblivious indexing to re-project each enumerated row to its original row.

In short, the STP first learns only the key columns of both sides (shuffled before revealed) and computes the intersection in the clear. Then, the result is secret-shared back out to the untrusted parties, and intersecting values are linked back to their original, unprojected rows. To see why this is secure, notice that the STP doesn't learn anything beyond the key row that they were allowed to learn, and all other parties receive just the join output. [14].

Hybrid aggregation can be used if the group-by column contains an STP. The steps are as follows:

1. Parties obliviously shuffle the input relations.
2. Parties send only the group-by column to the STP.
3. STP enumerates the group-by-column relations it receives, assigning an ID to each row.
4. STP sorts by the group-by column, putting all rows with the same group-by key next to one another.
5. STP computes an equality flag for each row, specifying if its key it equal to the one before it in the sorted relation.
6. STP secret-shares the equality flags to untrusted parties and reveals the indexes in the clear.
7. Parties perform oblivious scan over the results, obliviously aggregating the previous value to the current if the equality flag is true.

In short, the STP first learns the group-by column (shuffled before revealed), then enumerates the revealed keys. Locally, the STP shuffles by the group-by column, then computes equality flags for each row, true iff the row has the same group-by key as its predecessor in the sorted table. It then secret-shares the indexes and equality flags to the untrusted parties. The untrusted parties then obliviously scan and aggregate values according to the equality flag [14].

5. OBLIVIOUS SECURE ENCLAVES

Secure hardware may enable us to offload more MPC computation into a trusted setting as well. There is an emerging class of hardware primitives called secure enclaves that provide three security properties: isolate execution, sealing, and remote attestation [19]. Enclaves are being developed at Intel, AMD, and Apple, but all provide similar security guarantees [7]. *Isolated execution* means that an enclave has access to a subset of memory that no other process can access, not even the OS. *Sealing* means that the enclave can encrypt and decrypt data in such a way that only the enclave can interact with it undetectably. *Remote attestation* means that the enclave can prove that the code is running securely in the device. Enclaves, however, do not provide access pattern leakage attacks or side channel attacks. By monitoring the number of page accesses or the amount of power consumed by the enclave, attackers can still discern information about what is being computed over. Side-channel attacks are out of scope for this discussion; however, access pattern attacks aren't.

Systems have been develpoed that take advantage of these enclaves to further enable MPC by developing a new host of oblivious execution primitives for query execution [19]. In particular, these primitives protect against access pattern attacks. Contrary to the semi-honest security model we've interacted with thus far, we now assume adversary that can control the entire software stack, including control over network traffic, the operating system, and untrusted memory accesses. However, the adversary cannot compromise the enclave itself, nor any keys that the enclave holds. The primary kind of attack we want to protect against are access pattern attacks; to do so, we develop the notion of an oblivious primitive.

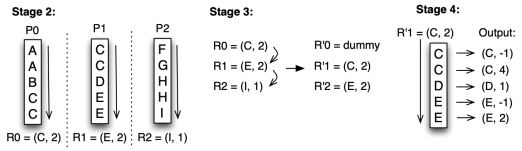

Figure 5: Oblivious aggregation

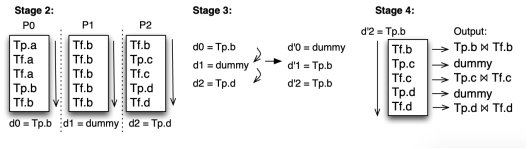

Figure 6: Oblivious join

5.1 Oblivious relational operators

The oblivious operators we explore rely chiefly on inter-machine oblivious sorting, which allows a set of machines to sort a table by some key without revealing anything about their data. Given a block of oblivious memory, we can split our data into block-sized chunks and quicksort each one, then run bitonic sort over each block in an enclave to recover the fully sorted result. We can further optimize this by using column sort, which preserves the *balance* of the partitions, preserving leakage of input table sizes. Given oblivious sorting, we can construct oblivious filters, aggregations, and joins in enclaves.

Oblivious filters are easy to implement; tag each row that should be included with a '0' and a '1' otherwise, then run oblivious sort by this new column. Then, remove all of the '1' rows.

Oblivious aggregations are more complicated to implement. The steps are as follows:

1. Sort by the grouping attribute to co-locate all elements that should be in the same group.
2. Naively scanning and aggregating can leak the number of values in each group. So instead, we repartition so that each worker holds a contiguous set of the total table.
3. Then, each worker outputs the statistic of the last row it holds and sends it to the next worker, called a global partial aggregate.
4. Upon receiving this value, each worker can calculate the statistics for groups that spill over from the previous worker and that are totally contained in this worker's partition.
5. We finish by filtering out dummy records.

The diagram in Figure 5.1 explains this data flow.

Oblivous sort-merge joins are even more complicated to implement. We describe an oblivious join that works when one of the tables is being joined by a primary key, then show an extension to arbitrary equi-joins. The steps are as follows:

1. We proceed in a similar fashion as in oblivious aggregations by first unioning both the left table, T_l, and right table, T_r, and then sorting by the join key. We assume that we are joining on T_l's primary key.
2. We repartition so that each worker holds a contiguous set of the total unioned sorted table.
3. Then, each worker outputs the last occurrence of a row from T_l, if it exists, to the next worker.
4. Upon receiving this value, each worker can calculate all of the rows that are supposed to join on the row from the previous worker, and all rows that are totally contained within this worker.
5. We finish by filtering out dummy records.

The diagram in Figure 5.1 explains this data flow. Extending this to arbitrary equi-joins is as easy as outputting the last set of entries that share a common join key from T_l instead of just the one record.

Using these operators, Opaque then runs cost-based query optimization to choose an optimal query plan. Moreover, the system allows users to specify the relative sensitivity of tables to allow more work to be done in the clear. Both of these optimizations harken back to prior discussion.

6. OTHER SOLUTIONS

MPC is far from the only solution to the problem is big data analytics over federated databases. Another popular solution is fully homomorphically encrypted databases, which leverage homomorphic encryption to answer queries over encrypted data. This flips the script on MPC by obscuring the data itself, rather than the computation over the data. Research has been done to make certain queries faster in homomorphically encrypted databases [5], sometimes with slight information leakage [8]. However, FHE don't hide access patterns, nor do they benefit from the same class of optimizations as MPC does since all of the data is already encrypted. While it can scale very nicely with the number of parties, key management can become arduous, and it incurs a hefty slowdown on large datasets. Such databases include CryptDB, BlindSeer, Monomi, AlwaysEncrypted, and Seabed. We thank [19] for this list.

Research has also been done on relying on trusted hardware to execute computation. However, without work to make computation oblivious, this approach may still be vulnerable to access pattern attacks [19]. Such databases include Haven, VC3, TrustedDB, TDB, and GnatDb. We once again thank [19] for this list.

7. CONCLUSION

Scaling MPC to industry needs will be tricky. However, we've seen a host of MPC-aware optimizations that incorporate MPC into the data processing pipeline nicely and minimally in a move towards scalability. Query processing in the trusted setting has been deeply researched; however, in

the untrusted setting with garbled circuits and secret shares as new primitives, new optimizations abound.

MPC-aware query optimization techniques based somewhat in traditional query optimization allow us to minimize the amount of MPC used on a table level. MPC-aware information flow analysis allows us an even finer level of control over what data needs to be kept secure, and from whom. In the execution layer, fine-grained knowledge of which parties can know what data allow for increased offloading of execution. And finally, secure hardware allows us to define new, safe primitives for query evaluation in the MPC setting.

While MPC is not the only solution for big data analytics over federated databases, it is certainly an enticing option rife with opportunities for improvement. I hope this has been a productive primer into the world of MPC-aware query optimization!

8. REFERENCES

[1] M. Armbrust, Y. Huai, C. Liang, R. Xin, and M. Zaharia. Deep dive into spark sql's catalyst optimizer. In *https://www.databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html*, 2015.

[2] V. Attasena, J. Darmont, and N. Harbi. Secret sharing for cloud data security. In *The VLDB Journal manuscript*, 2017.

[3] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. Smcql: Secure querying for federated databases. In *Proceedings of the VLDB Endowment 10.6*, 2017.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: a framework for fast privacy-preserving computations. In *https://eprint.iacr.org/2008/289.pdf*, 2008.

[5] C. Bösch, H. Pieter Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. In *ACM Computing Surveys 47.2*, pages 18:1–18:51, 2014.

[6] K.-M. Chung and R. Pass. A simple oram. In *ASIACRYPT, pages 197-214*, 2013.

[7] V. Costan and S. Devedas. Intel sgx explained. In *MIT CSAIL*, 2016.

[8] Z. Espiritu. Time- and space-efficient aggregate range queries on encrypted databases. In *Brown University*, 2022.

[9] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[10] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP'07*, 2007.

[11] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th VLDB Conference*, 1994.

[12] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. In *https://eprint.iacr.org/2004/175.pdf*, 2004.

[13] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE S&P 2014*, 2014.

[14] N. Volgushev, M. Schwarzkopf, A. Lapets, M. Varia, and A. Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys)*, 2019.

[15] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, 2009.

[16] Y. Yu, M. Isard, D. Fettery, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[17] M. Zaharia, M. Chowdhury, S. Michael Franklin, J and. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud 2010*, 2010.

[18] S. Zahur and D. Evans. Obliv-c: A language for extensible data-oblivious computation. In *http://eprint.iacr.org/2015/1153*, 2015.

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