An Oblivious General-Purpose SQL Database for the Cloud

Paper # 320

Abstract

We present ObliDB, a secure SQL database for the public cloud that supports both transactional and analytics workloads and protects against access pattern leakage. With databases being a critical component in many applications, there is significant interest in outsourcing them securely. Hardware enclaves offer a strong practical foundation towards this goal by providing encryption and secure execution, but they still suffer from *access pattern leaks* that can reveal a great deal of information. The naïve way to address this issue—using generic Oblivious RAM (ORAM) primitives beneath a database—adds prohibitive overhead. Instead, ObliDB co-designs both its data structures (e.g., oblivious B+ trees) and query operators to accelerate SQL processing, giving up to 329× speedup over naïve ORAM. On analytics workloads, ObliDB ranges from competitive to 19× faster than systems designed *only* for analytics, such as Opaque, and comes within 2.6× of Spark SQL. Moreover, ObliDB also supports point queries, insertions, and deletions with latencies of 1–10ms, making it usable for transactional workloads too. To our knowledge, ObliDB is the first oblivious database that supports both transactional and analytic workloads.

1 Introduction

Relational databases are a linchpin of modern computer applications, ranging from low-volume services inside an enterprise to global applications like Facebook. With the advent of cloud computing, there is considerable interest in running databases securely in the cloud, protecting their sensitive content from both network attackers and insiders at the cloud provider (e.g., a hacker who breaches the provider's security [47]). Researchers have proposed approaches including property-preserving encryption [23, 36, 37], trusted hardware [4, 55], and algorithms to run specific computations securely [33, 48, 51], giving various tradeoffs between security, generality and performance.

One of the most promising practical approaches to increase security is hardware enclaves such as Intel SGX [15]. These enclaves provide an environment where a remotely verifiable piece of code can execute without interference from the OS, accessing a small amount of enclave memory and making upcalls to the the OS when needed. However, applications using enclaves to manage a large amount of data must still access it through the OS (e.g., to read new memory pages or access the disk), which makes them susceptible to access pattern attacks. For database workloads in particular, access patterns can reveal a great deal of information even when the data is encrypted [26, 34, 52]. Some recent systems, such as Opaque [55] and Cipherbase [4], have proposed oblivious execution schemes that do not reveal access patterns, but these schemes are limited to *analytics* workloads that scan entire tables to answer a query. Specifically, both systems use oblivious sort operators that sort all the data. These systems would not be efficient for more general database workloads that also include transaction processing (point queries and updates to just a few records)—one of the most common use cases for databases.

This paper presents ObliDB, an oblivious SQL database that supports both transactional and analytical processing using hardware enclaves. ObliDB goes beyond prior work by providing new *storage methods* (e.g., oblivious B+ trees) and new *operators* that can efficiently support transactional queries, in addition to analytics on the same data. These contributions make ObliDB the first oblivious system to support—and achieve acceptable performance on—both transactional and analytics workloads.

Unlike previous systems, ObliDB supports two *storage methods* for its tables: a linear one where the table is encrypted as a contiguous file and always scanned (as in Opaque and Cipherbase), as well as an *oblivious B+ tree* for efficient indexed access. Each table can be stored using either one or both methods. The key idea in ObliDB's oblivious B+ trees is to use techniques from Oblivious RAM (ORAM) [46] to support fast lookups and updates to just part of the database. However, naïvely applying ORAM under a standard database B+ tree (e.g., substituting a generic ORAM library like ZeroTrace [40] for each memory access) would give prohibitively high overheads. Instead, ObliDB designs an oblivious B+ tree tailored for ORAM, and operators on it that perform orders of magnitude better than a naïve implementation. For example, an oblivious system needs every tree operation (e.g., node insertion) to take the *worst-case* time, or an attacker would observe information about the inserted node; however, standard tree implementations do not aim to minimize this worst case time. ObliDB's B+ tree addresses this and other performance challenges.

The new storage methods in ObliDB also require new implementations of SQL operators such as selections and joins. ObliDB provides multiple versions of each operator based on the input and output storage methods, as well as other properties, such as data sizes. As with our B+ tree, we carefully design the operators to be oblivious, allowing them to outperform a naïve implementation over ORAM.

ObliDB also provides a query optimizer to automatically select the best operator implementations for a query at runtime when there are multiple options.

Together, these features let ObliDB support a wide range of queries efficiently without leaking any information beyond intermediate result sizes and the chosen query plan—the same security level as Opaque's oblivious mode.[1] ObliDB supports selections, aggregations and joins similar to other analytics systems [4, 55], as well as efficient point lookups, insertions, deletions and updates.

We evaluate a prototype of ObliDB on diverse applications and find that ObliDB performs acceptably, both in absolute terms and relative to other secure systems. We first compare ObliDB to a baseline implementation where a database index is generically modified to run over ORAM, and show that ObliDB outperforms it by up to 329×. For analytics workloads, we compare ObliDB to Opaque on the Big Data Benchark [3] and find that ObliDB is competitive with Opaque on most queries, but can also outperform Opaque by 19× on queries that can leverage indexes. ObliDB also comes within 2.6× of Spark SQL [6], which provides no security guarantees. For transactional workloads, ObliDB outperforms the recent Sophos encrypted search scheme that does not hide access patterns [11] by over 22×. Moreover, point inserts, deletions and selects on a 1M row dataset take 3.6–9.4ms, which is less than the network latency in many applications. Finally, we show that the choice of oblivious operators available in ObliDB enables meaningful optimizations during the query planning process.

2 Overview

This section summarizes the functionality and architecture of ObliDB, its threat model, and its security properties.

2.1 Threat Model

We assume an attacker with full control of the operating system (OS) running ObliDB, including the power to examine and modify untrusted memory, network communication, and communication between the processor and enclave. Moreover, the attacker can observe access patterns to trusted memory and maliciously interrupt the execution of an enclave. We note that an OS-level attacker can always launch an indefinite denial of service attack against an enclave, but such an attack does not compromise privacy, so we do not attempt to prevent it.

We assume the security of the trusted hardware platform in that the enclave hides the contents of its protected memory pages from an attacker with control of the operating system. While some side-channel attacks based on abusing page faults and branching history have been demonstrated against Intel SGX [27,52], a number of standard mitigations exist to handle these attacks [38, 42–44], and other hardware enclave designs avoid the pitfalls that leave SGX vulnerable [16, 28, 30].

Furthermore, we also assume a secure channel exists through which a user can send messages to the enclave. A client can establish such a connection, e.g., through TLS.

2.2 Security Goals

We focus on protecting against access pattern attacks [26]. Although encryption can hide the exact text of data, knowledge about what parts of memory have been accessed and when can reveal a great deal of sensitive information. For example, consider an alphabetical list of student records. Knowledge of which position in a table has been read directly reveals the subject of a query. Another example is a database of information on diseases [55]: correlating the frequency at which particular rows are accessed with the known incidence of diseases can reveal the malady from which a patient suffers.

Our Guarantees. Queries in ObliDB leak only the sizes of tables involved, including sizes of intermediate and result tables, as well as the query plan used. This security level is the same as Opaque's oblivious mode [55]. ObliDB additionally features a padding mode where all tables are padded to some chosen size. Further details regarding how we achieve these leakage properties appear in Sections 4 and 5, which discusses the implementation of each operator. Data at rest outside the enclave is encrypted and MACed and leaks only its size. We do not make an effort to hide the number of tables in a database or which table(s) a particular query accesses.

We additionally make the integrity guarantee that ObliDB catches and reports any tampering with data by the malicious OS. We use a series of checks and safeguards to protect against arbitrary tampering within rows of a table, addition/removal of rows, shuffling of the contents of a table, or rollbacks to a previous system state. We discuss these protections in Section 4.

We can formally model our guarantees with a *simulator* that, given only query plans and table sizes, produces an output distributed identically to the view of an adversary. Since the simulator *only* sees what we intend to leak, the adversary cannot have learned any additional information from its interaction with our system. In this model, the theorem statement of Opaque [55] also applies to ObliDB.

2.3 ObliDB Architecture Overview

Figure 1 shows an overview of the ObliDB architecture. ObliDB consists of a trusted code base inside an enclave that provides an interface for users to create, modify, and query tables. ObliDB supports two *storage methods* for each table: Linear and Indexed. It stores tables, encrypted, in unprotected memory and obliviously accesses them as needed by the various supported operators. The Indexed

[1] ObliDB also supports padding intermediate and final results to a fixed size, similar to Opaque's pad mode, if desired.

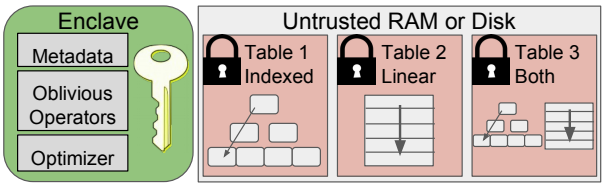

Figure 1: ObliDB provides an interface to a secure enclave with control over encrypted tables stored in untrusted memory. It stores tables either as an oblivious B+ tree index, a linear scan data structure, or both to perform efficient oblivious queries.

method consists of an ORAM with a B+ tree stored inside, whereas the Linear method requires scanning the whole table on each query to ensure obliviousness.

ObliDB supports oblivious versions of the SQL operators `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `GROUP BY` and `JOIN` as well as the aggregates `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`. Each operator is implemented for both storage methods. Finally, ObliDB includes a query optimizer that can choose the right operator implementation for each query. For example, for selection queries, the optimizer first determines the size of the selection and then executes the best-performing `SELECT` algorithm for the data to be returned. Choices include algorithms that take advantage of the possibility of caching small results inside the enclave or quickly handling very large return sets by making a copy of the original table and returning it whole with the few missing rows obliviously erased. ObliDB also provides a padding mode that pads all tables to a chosen size. In this mode, it does not need perform optimizations based interemediate on data size.

2.4 Limitations

ObliDB has two limitations that we wish to point out. First, the current system is not distributed: ObliDB is designed to run on one node, storing data in its memory and disks. In practice, however, many important database workloads fit on one node, as evidence by the popularity of cloud services that run single-node databases such as MySQL and Postgres (e.g., Amazon RDS [2]). Ideas from ObliDB could also be used in distributed systems in several ways—for example, implementing ObliDB's oblivious B+ tree over S3 to store large amounts of data, or combining ObliDB with cross-machine oblivious sort operators from Opaque [55] to perform distributed queries.

Second, despite ObliDB's security guarantees, an application interacting with it can leak additional information and must therefore be secured as well. For example, if a web application using ObliDB makes a second query to a database based on the results of a first query, observing the size of the response to the second query—or even the fact that the query occurred—may leak additional information about the first query. This limitation would exist with any oblivious database system, so application developers need to consider it in their design process.

3 Background

In this section we give a basic overview of hardware enclaves and ORAM, the primary tools used in ObliDB, providing only sufficient detail for the later sections.

3.1 Enclaves and Intel SGX

A *hardware enclave* provides developers with the abstraction of a secure portion of the processor that can verifiably run a trusted code base (TCB) and protect its limited memory from a malicious or compromised OS [1, 15]. The hardware handles the process of entering and exiting an enclave and hiding the activity of the enclave while non-enclave code runs. Enclave code invariably requires access to OS resources, so developers specify an interface between the enclave and the OS. In SGX, the platform we use in our implementation, the functions made available by this interface are called *OCALLs* and *ECALLs*. OCALLs are made from inside the enclave to the OS, usually for procedures requiring resources managed by the OS, such as file access. ECALLs allow code outside the TCB to call the enclave to execute trusted code.

An enclave proves that it runs an untampered version of the desired code through a mechanism named *attestation*. Attestation involves an enclave providing a hash of its initial state which a client compares with the expected value of the hash and rejects if there is any evidence of a corrupted or altered program.

The most significant feature of enclaves for our purposes concerns the protection of memory. An enclave gives developers a small Enclave Page Cache (EPC), a memory region hidden from the OS and cleared whenever execution enters or exits an enclave. In this memory, the trusted code can keep secrets from a malicious OS who otherwise controls the machine. SGX provides approximately 90MB of EPC. Beyond using EPC, code in the enclave has to call the OS to access other memory pages.

3.2 ORAM

Oblivious RAM (ORAM), a cryptographic primitive first proposed by Goldreich and Ostrovsky [24], hides access patterns to data in untrusted memory. In the traditional ORAM setting, a small trusted processor uses a larger memory over a bus where an adversary examines communications. Merely encrypting data over the bus still reveals the access patterns to the data being requested and can reveal private information about the data or the queries on it [26]. ORAM goes further and shuffles the locations of blocks in memory so repeated accesses to the same block and other patterns are hidden from the adversary. ORAM guarantees that any two sets of access patterns of the same length are indistinguishable from each other. ORAM security is formally defined as follows:

Definition 1 (ORAM Security [46]). *Let* $\overrightarrow{y} := ((op_M, a_M, data_M),...,(op_1, a_1, data_1))$ *denote a data re-*

quest sequence of length M, where each $\mathbf{op}_i$ *denotes a* **read(a_i)** *or a* **write(a_i, data)** *operation. Specifically,* $\mathbf{a}_i$ *denotes the identifier of the block being read or written, and* $\mathbf{data}_i$ *denotes the data being written. Index* 1 *corresponds to the most recent load/store and index M corresponds to the oldest load/store operation.*

Let $A(\overrightarrow{y})$ *denote the (possibly randomized) sequence of accesses to the untrusted storage given the sequence of data requests* $\overrightarrow{y}$*. An ORAM construction is said to be secure if:*

1. *For any two data request sequences* $\overrightarrow{y}$ *and* $\overrightarrow{z}$ *of the the same length, their access patterns* $A(\overrightarrow{y})$ *and* $A(\overrightarrow{z})$ *are computationally indistinguishable by anyone but the client ORAM controller.*
2. *The ORAM construction is correct in the sense that it returns on input* $\overrightarrow{y}$ *data that is consistent with* $\overrightarrow{y}$ *with probability* $\geq 1 - negl(|\overrightarrow{y}|)$*, i.e., the ORAM may fail with probability* $negl(|\overrightarrow{y}|)$*.*

The scope of ORAM's security guarantees create important consequences for oblivious data structures or algorithms built on it. Specifically, *ORAM only guarantees indistinguishability for access patterns of the same length*, so algorithms using ORAM must always make the same number of memory accesses or risk leaking information.

Although other, older schemes have recently received attention due their practical efficiency in certain practical parameter settings [53], the most efficient ORAM scheme known is the Path ORAM [46]. Path ORAM belongs to a family of schemes known as tree-based ORAMs, which operate by storing the blocks of the oblivious memory in a tree structure. Each block is associated with a leaf in the tree in a position map that guarantees the block will be found somewhere on the path to that leaf. An access to the ORAM involves reading a path down the tree from the root to the leaf corresponding to the desired block. After retrieving the desired block, a second pass is made on the same path where each block is re-encrypted with new randomness and the retrieved block is assigned a new leaf, remaining stored in a small "stash" if the path does not allow space for it to be written back on the path to its new assigned leaf. Although it is not always necessary in practice, the position map holding the assigned leaves for each block of the ORAM can be recursively stored in its own ORAM to reduce the trusted processor memory required by this scheme to a constant.

4 ObliDB's Storage Methods

ObliDB can store database tables via two methods — Linear and Indexed — or combine both. We currently have system administrators decide which storage method(s) to use for each table, a decision easily made based on the kinds of queries that are expected to be run on the data. ObliDB creates tables with an initial maximum capacity that can be increased later by copying to a new, larger table. Since tables are stored in unprotected memory, ObliDB independently encrypts and MACs every block with a symmetric key generated inside the enclave. For both storage methods, it stores each row in one block and reserves the first byte of each block as a flag to indicate whether that block contains a row or is empty.

4.1 Linear Storage Method

The Linear storage method simply stores rows in a series of adjacent blocks with no additional mechanism to ensure obliviousness of memory accesses. This constitutes a "trivial" ORAM where every read or write to the table must involve accesses to every block in order to maintain obliviousness of access patterns. As such, operators acting on these tables, as will be seen in Section 5, involve a series of linear scans over the entire table. This performs best with small tables, tables where operations will typically require returning large swaths of the table, or aggregates that involve reading most or all of the table regardless of the need for obliviousness. The challenge in designing algorithms for this storage method lies in using the limited space of the enclave effectively to reduce the number of scans and data processing operations involved in each operator.

4.2 Indexed Storage Method

The Indexed storage method makes use of both an ORAM and a B+ tree in order to provide better performance without losing obliviousness for large data sets. It consists of an ORAM that holds a B+ tree where the actual data of the table resides, with each node of the B+ tree corresponding to one block of the ORAM. The security properties of ORAM guarantee that two access transcripts of the same length will be indistinguishable from each other, but we must ensure that the total number of accesses or the timing gaps between accesses do not leak any private data. The property of B+ trees that all data resides in the leaves of the tree, always at the same depth, means that any search in the tree will make the same number of accesses to intermediate nodes before finding the desired data. Using a different data structure that does not exhibit such a property would compromise the obliviousness of operators in our Indexed storage method.

The main contributions of our oblivious B+ tree are:

1. Insights into new tradeoffs in parameter selection for the ORAM setting
2. Optimizing for the necessity induced by obliviousness of every operation having *worst-case* runtime
3. Managing dynamic memory over ORAM.

Parameter Tradeoffs. The key insight in making this method efficient is that an ORAM underneath the B+ tree alters the usual performance tradeoffs inherent in B+ trees

in unexpected ways. For example, in a typical B+ tree, the choice of maximum degree of nodes in the tree depends on a balance between achieving low depth overall and not having too many pointers to examine and update within a single node. ORAM upsets the balance on which this intuition relies by increasing the cost of following or updating pointers between nodes(each of which requires an ORAM lookup) to the point where many operations scanning child keys within a single node become almost free by comparison to following a pointer. As such, the degree of a node can become even bigger than usual well before the cost of operations within a node reaches parity with the cost of an ORAM operation. On the other hand, ORAM presents another constraint too: increasing the degree beyond a certain point requires ORAM blocks to become larger, reducing the performance of the underlying ORAM. We compromise by making the degree of our trees as high as possible without artificially inflating the size of ORAM blocks to accommodate pointers to additional children.

Optimizing Worst-Case Performance. Since we wish to preserve obliviousness in all queries, every B+ tree operation takes on its worst-case running time every time it runs, and the costs of usually uncommon splits and merges of nodes must be paid on every insertion or deletion. To help mitigate the impact of this average case to worst case conversion incurred by obliviousness, ObliDB reduces the number of writes needed to the ORAM for each B+ tree operation, operating on a lazy write-back principle where it does not write changes to the ORAM until the last possible moment.

Another example of where ORAM changes implementation decisions for oblivious B+ trees in surprising ways regards the case of parent pointers. A normal B+ tree may have pointers in each node to its parent as a low-cost solution to avoid a search down the tree to find a node's parent. This optimization even appears useful at first glance because we want to minimize the worst case time for each operation, and a parent pointer helps expedite the process of finding parents for the purposes of merging or splitting nodes, saving a handful of ORAM reads on every operation. However, each time a tree splits or merges a node, all the children of nodes involved need to have their parent pointers updated, a very slow process in the regime where every node requires an ORAM write to update.

Memory Management. The structure of a B+ tree changes dynamically as rows are added and removed from a database, so our B+ tree implementation must use some form of dynamic memory management and pointers between nodes in the tree. We implement equivalents of malloc, free and the pointer dereference operator for our ORAM. Our memory allocator consists of an array of flags that we set if the corresponding block is in use and unset if it is not. This increases the protected memory needed over the ORAM's position map by 20% but does not represent a dramatic increase in memory requirements over the total space needed by ObliDB for the position map, ORAM stash, and other elements of system state recording the names, sizes, and types of existing tables.

Method	Linear	Index	Combined
Space	N	~$4N$	~$5N$
Point Read	$O(N)$	$O(log^2 N)$	$O(log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insertion	$O(1)$	$O(log^4 N)$	$O(log^4 N)$
Deletion	$O(N)$	$O(log^4 N)$	O(N)

Figure 2: Asymptotic performance of each storage method.

4.3 Complexity

Figure 2 compares the asymptotic operations of standard read, insertion, and deletion operations on each table type. The Indexed method performs best on small reads that access one or a few rows of a table, whereas queries which expect to return large segments of a table should use the Linear method, which performs faster than a linear scan over the contents of an index despite equal asymptotic runtimes. Using both storage methods, while incurring the cost of both for insertions and deletions, proves effective when queries of diverse selectivities run on the same data.

4.4 Data Integrity

Although encryption and oblivious data structures/algorithms ensure the privacy of data in ObliDB, additional protections make certain that an attacker does not tamper with data. Such tampering could take the form of editing within rows of a table, addition/removal of rows, shuffling the contents of a table, or rollbacks to an old system state. ObliDB protects against such attacks and reports any attempt to tamper with data.

ObliDB MACs and encrypts every block of data stored outside the enclave, preventing the OS from modifying rows or adding new rows to tables. This leaves the possibility of duplicating/removing rows, shuffling rows, or rolling back the system state. Included in each block of MACed data is a record of which row the block contains and its current "revision number," a copy of which ObliDB also stores inside the enclave. Any attempt to duplicate, shuffle, or remove rows within a data structure will be caught when an operator discovers that the row number of data it has requested does not exist or does not correspond to that which it has received. Spoofing a fake revision number requires either breaking the security of the MACs used or breaking the security of the enclave to modify the stored copy, neither of which lies within the power of an attacker in our model. Rollbacks of system state are caught when the revision numbers of rows in a table do not match the last revision numbers for those rows recorded in the enclave. These lightweight protections suffice to discover and block any malicious

tampering with data in ObliDB.

5 Oblivious Operators

In this section we describe the various oblivious operator algorithms used in ObliDB. ObliDB provides support for a large subset of SQL, including insertions, updates, deletions, joins, aggregates (count, sum, max, min, average), groupings, and selection with conditions composed of arbitrary logical combinations of equality or range queries. Moreover, depending on known information about the size of a response to a query, ObliDB can choose which algorithm to use in order to maximize performance in each situation. We will begin by discussing algorithms for the Linear storage method and then discuss the modifications or entirely different solutions used for the Indexed method. Each operation will be accompanied by a security argument.

Optimizer. For queries where ObliDB provides multiple possible algorithms, we include an optimizer that picks the best algorithm to use for the given query based on the data to be returned by the query (whose characteristics we determine via a fast initial scan). The optimizer decides which variant of an operator to use based on a pre-calculated table of rules that delineate the boundaries of the regime where each algorithm performs best. For maximum flexibility, users can also manually choose a particular algorithm, e.g. if a query runs in padding mode without information on query selectivity.

The following notation will be used in subsequent paragraphs: the table being returned will be referred to as R, and the table being selected from will be referred to as T. The number of rows in R is represented by r, the number of rows in T is N. r' and N' represent the number of blocks in the data structures holding R and T, respectively.

5.1 Linear Storage Method

Insert, Update, Delete. Insertions, updates, and deletions involve at most one pass over the table, during which unaffected blocks receive a dummy write (overwriting a row with the data it already held, re-encrypted) and affected blocks are written to as follows:

Insertion: ObliDB offers two options for insertions. First, it can conduct a linear scan, making dummy writes on each row except the first unused block, where it makes a real write. In tables with few deletions, a fast insertion algorithm recalls the last row where an insert occurred and inserts directly into the next row, skipping the scan.

Deletion: any row matching the deletion criteria will be marked as unused and overwritten with dummy data. Deletions and updates support arbitrary conditions, similar to selections, so any logical combination of conditions on equality or inequality of entries in a row is acceptable.

Update: any row matching the update criteria will have its contents updated instead of a dummy write.

All of the above operations leak nothing about the parameters to the query being executed or the data being operated on except the sizes of the data structures involved because they consist of one linear scan over a table where each encrypted block is read and then written with a fresh encryption. Fast insertion also leaks no additional information beyond the sizes of tables because the access pattern of the insert does not depend at all on the content of the data except on the number of insertions made, which our adversary can already learn by observing the sizes of tables over time.

Select. Note that satisfying a `SELECT` query via a straightforward scan that copies each row matching the given criteria into an output table does not provide obliviousness despite touching every row in the table. Such an approach still leaks which rows we include in the output because an attacker can watch the output table and take note of which points in the scan coincide with growth in the output table. Our Select algorithm begins by scanning once over the desired table and counting the number of rows that are to be selected. This step leaks only the size of the table T. Then, based on the size of the output set and whether the selected rows form one continuous block in the table or not, it executes one of several strategies:

Naïve: included as a baseline for comparison, the naïve oblivious algorithm mirrors a straightforward translation of a non-oblivious `SELECT` to an oblivious one via an ORAM. After examining each row, it executes an ORAM operation. If the examined row is to be included in the output, it makes a write. If not, it makes a dummy read (reading an arbitrary block). After completing the scan of the input table, it copies the contents of the ORAM to a linear storage format and returns it.

Our techniques to improve upon this baseline consist of finding the right balance between using data structures inside the enclave to remove the need for an ORAM and making multiple fast passes over data. These ideas constitute the guiding principle in designing our remaining `SELECT` algorithms and choosing between them.

Continuous: Should the rows selected form one continuous section of the data stored in the table, a common special case, ObliDB employs a strategy that requires only one additional pass over the table. First, it creates table R with r rows. Then, for the ith row in table T, if that row should be in the output, it writes the row to position $i \bmod r$ of R. If not, it makes a dummy write. Since the rows that need to be included in R make up one continuous segment of T, this procedure results in exactly the selected rows appearing in R.

In addition to the sizes of tables T and R, the fact that ObliDB chooses this algorithm over one of the other options leaks the fact that the result set is drawn from a continuous set of rows in the table. Users concerned about this additional leakage could disable this option

and use one of the other options with no reduction in supported functionality. The execution of the algorithm itself is oblivious, however, because the memory access pattern is fixed: at each step, the algorithm reads the next row of T and then writes to the next row of R.

Small: In the case where r is small, that is, where all the rows of table R only require a few times the space available in the enclave, a selection strategy that makes multiple fast passes over the data proves effective. We take multiple passes over table T, each time storing any selected rows into a buffer in the enclave and keeping track of the index of the last checked row. Each time the buffer fills, its contents are written to R after that pass over T. Although this strategy could result in a number of passes linear in the size of R, it proves effective for small r, as demonstrated in Section 7.

This algorithm leaks only the sizes of tables T and R because every pass over the data consists only of reads to each row of the table and the number of passes reveals only how many times the output set will fill the enclave, a number that can be calculated from the size of R, which we reveal anyway.

Large: If table R contains almost every row of table T, we create R as a copy of T and then make one pass over R where each unselected row is marked unused and each selected row receives a dummy write.

The copy operation reveals no additional information about T or R because it can be carried out by the OS with no input from the enclave or a user. The process of clearing unselected rows involves a read followed by a write to each block of the table, so it also reveals no information beyond the size of T. This algorithm, in fact, does not even reveal the size of the output set R because we pad the data structure to the size of T.

Hash: In the case that none of the preceding special-case algorithms apply, ObliDB uses the following generalization of the continuous strategy. We wish to apply the technique used for continuous data on data that may be arbitrarily spread throughout T, not just in one continuous block. Our approach is to resort to a hashing-based solution. For the ith row in T, if the row is to be included in the output, we write the content of the row to the $h(i)$th position in R, for some hash function h.

The algorithm as stated above does not exactly represent how ObliDB works because we need a few changes in order to ensure, first, that we properly handle hash collisions to ensure correctness, and, second, that we maintain obliviousness in handling collisions. In order to maintain obliviousness, every real or dummy write to R must involve the same number of accesses to memory. This means that if any write resolves in a collision, every write must make as many memory accesses as in the case of a collision. Following the guidance of Azar et al [9], we use double hashing and have a fixed-depth list of 5 slots for each position in R. This means that for each block in T, there will be 10 accesses to R, 5 for each of the two hash functions.

The modifications above ensure that data access patterns are fixed regardless of the data in the table and which rows the query selects. Since the hash is taken over the index of the row in the data structure and not over the actual contents of a row, information about the data itself cannot be leaked by access patterns when rows are written to R. As such, only the sizes of T and R leak. The selection strategy also leaks, but this information can be deduced just from knowledge of the sizes of T and R and therefore leaks no additional information.

Aggregates & Group By. A baseline solution to aggregation queries performs very poorly, but we can compute aggregates far faster than selection. An aggregate over a whole table or some selected subset of a table requires only one pass over the whole table where we calculate the aggregate cumulatively based on the data in each row. A naïve approach uses an ORAM to keep track of the aggregate and needs to access it for every row, causing an unnecessary slowdown. We achieve better performance by keeping the aggregate statistic inside the enclave and avoiding the ORAM overhead. Since the memory access pattern of this operation always involves sequential reads of each block in the data structure, nothing leaks from this operation beyond the size of table T.

We handle groupings similarly to aggregates without groupings, except we keep an array inside the enclave that keeps track of the aggregate for each group where a naïve solution would check an entire array via oram for each row of table T. The method for determining which group each row belongs to is handled differently for low and high-cardinality aggregation:

Low-Cardinality: In the low-cardinality setting, we make a linear scan over known groups in order to check for a match. If we find no match, we create a new group.

High-Cardinality: Linearly scanning over all known groups becomes prohibitively expensive as the number of groups becomes larger, so high-cardinality groupings employ a hash table where each group's value is hashed and inserted into a hash table held in the enclave. Each row scanned is hashed and checked against the table. If there is a match, then the row under examination corresponds to a known group referenced in the table, and if not, then the current row is added as a new group.

Join. We implement joins for the Linear storage method as a variant of the standard hash join algorithm [19]. We refer to the two tables being joined as T_1 and T_2. We make a hash table out of as many rows of T_1 as will fit in the enclave and then hash the variable to be joined from each row of T_2 to check for matches. This process repeats until reaching the end of T_1 . After each check, a row is written to the next block of an output table. If there is a match, the

joined row is written. If not, a dummy row is written to the table at that position. We reveal the sizes of the tables T_1 and T_2, but not the size of the output table, which is padded to a parameter representing the maximum possible size by dummy rows (which can be less than $|T_1| * |T_2|$ if desired). Since each comparison between the tables always results in one write to the output structure, the memory access pattern of this algorithm is oblivious.

5.2 Indexed Storage Method

Operations for the Indexed storage method behave similarly to the Linear method, but all operations take place over the ORAM and B+ tree data structure described in Section 4. The important difference between the two lies in the fact that the index can restrict a search to a particular relevant area of a table without having to scan every row to maintain obliviousness. The use of an index, however, comes with some security ramifications. If the rows returned by a query are not continuous, the leakage also includes the size of the segment of the database scanned in the index. For example, supposing that there is one student named Fred in a table of students indexed by student IDs, the query `SELECT * FROM students WHERE NAME = ''Fred'' AND ID > 50 and ID < 60` leaks not only that the size of the result set is 1 but also that 9 rows were scanned in the execution of the query. We consider this leakage to be included in the sizes of intermediate tables, as a query plan that selects a noncontinuous segment is equivalent to one which selects a continuous segment from an index and then selects a noncontinuous segment from the returned table. This leakage can be hidden by padding.

Insert, Update, Delete. Standard insertion and deletion operations for B+ trees leak information about the internal structure of the tree because they involve splitting and merging nodes when they reach fixed threshold numbers of children. Instead, our insertions and deletions pad the number of operations made on the underlying ORAM so no information can be leaked about the internal structure of the B+ tree. As discussed in Section 4, this means every B+ tree operation has worst-case running time and that design decisions involved in constructing the trees and operations on them differ from the traditional setting without ORAM. Updates on segments of a table behave similarly to the Linear storage method.

Select, Aggregates, & Group By. We handle selection, aggregation, and grouping algorithms as before, with the major difference that we use an index to find the part of a table to scan. The Large selection strategy does not apply to the Indexed storage method because the strategy of copying the whole table is not as applicable where a query aims at a small fraction of the table. The Indexed storage method can also be scanned linearly as a table using the Linear method would be, but this scan has overhead above Linear method scans because of the extra space required by ORAM. In practice, this overhead is less than $2\times$.

Join. Rows are always sorted by the index column in the leaves of the B+ tree, so it is possible to efficiently sort-merge join tables with indexes on the same column [19]. Tables T_1 and T_2 are scanned at the same time, and matching rows are placed in an output ORAM, as in the Linear storage method. Specifically, at each step, the next row of each of T_1 and T_2 is read. If the rows match, the pointer on the table T_2 advances and there is a write to the ORAM, and if they do not match, a dummy write takes place and the pointer on the table with the lesser value advances. This process proceeds until pointers reach the end of both tables. Obliviousness holds because each step of the algorithm consists of exactly one read to each of T_1 and T_2 and one write to an ORAM, and the total number of steps is only a function of the sizes of the tables involved.

5.3 Complexity

Prior systems that implement oblivious operators include Opaque [55], Cipherbase [4], and the Oblivious Query Processing algorithms of Arasu and Kaushik [5]. All three works focus on oblivious algorithms for analytic queries and only propose algorithms that involve scans over entire tables. The approach to this kind of operator typically involves a combination of oblivious sorts and filters. In contrast, our work uses new ideas to achieve similar functionalities for both storage methods, providing support for a broader set of general database use cases. Whereas sort and filter based approaches always have complexity $O(N)$ or $O(NlogN)$ in the size N of a table, ObliDB's solutions range in complexity from $O(1)$ to $O(N^2)$, but our optimizer picks the algorithms it expects will perform best in practice regardless of asymptotics.

6 Implementation

Our implementation includes the storage methods from Section 4 as well as the oblivious operator algorithms described in Section 5. It consists of over 14,000 lines of code of which approximately 10,000 are new and builds upon the Remote Attestation sample code provided with the SGX SDK [1] and the B+ tree implementation of [8], the latter of which was heavily edited in order to support our ORAM memory allocator. We used SGX SDK libraries for encryption, MACs, and hashing. We will make ObliDB open source and publicly available online.

We tuned ObliDB's parameters for the protected memory space provided by SGX. We chose a nonrecursive PATH ORAM [46] for our ORAM scheme. This ORAM can fit up to about 15 million rows before needing a second layer of recursion in order to fit in an SGX enclave, so ObliDB can handle realistic data sets without recursive ORAM. That said, ObliDB can be modified to use recur-

Table Name	Rows	Notes
CFPB	107,000	Customer complaints to the US Consumer Financial Protection Bureau [20].
USERVISITS	350,000	Server logs for many sites. Part of the Big Data Benchmark data set [3].
RANKINGS	360,000	URLs, PageRanks, and average visit durations for many sites. Part of the Big Data Benchmark data set [3].

Figure 3: Real data used in our evaluation and comparisons.

sive ORAM at a modest performance penalty for the Indexed storage method because our implementation allows easy swapping of ORAM schemes through a common interface. This means our choice of ORAM can easily be replaced to optimize the ORAM scheme as in [53].

7 Evaluation

We evaluate ObliDB on tables of up to 1.4 million rows, comparing to both prior work and a baseline implementation that naïvely modifies a database to use ORAM as well as existing private database systems. Real-world data sets used in our experiments appears in Figure 3. We also measure the overhead of ObliDB's padding mode and demonstrate the effectiveness of ObliDB's query optimizer as well as the efficacy of our storage methods in different situations through a series of microbenchmarks. We evaluated ObliDB on a desktop computer with an Intel Core i7-6700 CPU @3.4GHz and 8GB of RAM running Ubuntu 16.04.2 and the SGX Linux SDK version 1.9 [1].

We find that ObliDB dramatically outperforms a baseline implementation and can leverage its indexes to achieve order of magnitude performance improvements over previous private database systems. In particular, ObliDB matches Opaque [55] for scan-based queries but can outperform it by 18.8x when it uses an index. ObliDB also performs 22.6-24.6× faster than Sophos [11], a recent index-based searchable encryption scheme.

7.1 Comparison to Naïve ORAM Baseline

To evaluate the impact of ObliDB's specialized storage methods and operators, we also implemented a baseline database system that naïvely uses ORAM. Since a direct translation of the memory accesses of a non-oblivious data structure to an ORAM does not guarantee obliviousness (e.g., if different operations make different amounts of ORAM accesses), we modified several standard database data structures and algorithms as little as possible to achieve an oblivious version for our baseline, always erring on the side of stronger performance, the ultimate goal being to simulate the behavior of a generic conversion system that would render legacy code oblivious. Our baseline uses the same data structure as ObliDB for the Linear storage method but uses a naïve B+ tree implementation that does not take advantage of any of the ORAM-related optimizations discussed in Section 4. That is, it writes back to the ORAM every time part of the tree changes instead of waiting as long as possible to remove redundant writes and does not optimize parameters for the ORAM setting. Moreover, it keeps in each node a pointer to that node's parent, a shortcut that usually helps, but, as discussed in Section 4 damages oblivious performance because of the average case to worst case conversion that obliviousness induces. The baseline also uses the naïve varieties of operators as described in Section 5.

Figure 4 compares ObliDB to our naïve ORAM baseline. ObliDB achieves up to 29× speedup for `SELECT` queries and over 328× speedup for aggregates. `SELECT` queries over the Linear storage method enjoy much larger speedup over the baseline than index queries because an oblivious B+ tree lookup takes most of time in the indexed `SELECT` queries. We used the same algorithm for this lookup in both the baseline and actual implementations because a naïve application of ORAM to a B+ tree does not yield an oblivious B+ tree, as described in Section 4.

The largest speedups appear in aggregation queries, where ObliDB gains two orders of magnitude in performance. This arises from the need to hide data structures that keep statistics for each group without revealing when a row does not match with any known groups and needs to begin a new group. The possibility of this occurrence forces, in the naïve algorithm, an access to each group's data for each row. With a high system-wide maximum number of groups, such a query cannot complete within a reasonable time frame and may take well over the 1,000 seconds at which we cut off our experiments. The aggregation query over the CFPB table completes in a shorter period of time because we used our prior knowledge of the number of banks to set the maximum number of groups to a lower threshold (200, in this case).

7.2 Comparison to Opaque

Figure 5 compares ObliDB with Opaque's oblivious mode [55] and Spark SQL [6] on queries 1-3 of the Big Data Benchmark [3] (the same queries used by Opaque) on tables of 360,000 and 350,000 rows. We omit the benchmark's query 4 as neither ObliDB nor Opaque support the external scripts it requires. Opaque also uses an SGX enclave and can be configured in either "encryption" mode, which leaks access patterns but offers performance close to Spark SQL and "oblivious" mode that hides access patterns to data, but by means different from ours. Spark SQL provides no security guarantees. We run both Opaque and Spark SQL in single node configuration.

We began by configuring ObliDB to use only the Linear storage method, as Opaque does, and found that ObliDB

Data Set	Query	ObliDB	Baseline	Speedup
	Linear Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	1.192s	34.79s	29.2×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	2.434s	46.33s	19.0×
	Index Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	0.472s	0.678s	1.4×
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2017-08-17` (point query)	0.0027s	0.0033s	1.5×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	0.082s	0.107s	1.3×
	Index Insertion/Deletion			
CFPB	`INSERT INTO CFPB (Complaint_id, Product, Issue, Date_received, Company, Timely_response, Consumer_disputed) VALUES (4242, "Credit Card", "Rewards", 2017-09-01, "Bank of America", "Yes", "No")`	0.011s	0.708s	64.4×
CFPB	`DELETE FROM CFPB WHERE Bank="Bank of America" AND Date_Received=2017-08-17` (point query)	0.015s	0.220s	15.0×
	Aggregates and Joins			
CFPB	`SELECT COUNT(*) FROM CFPB WHERE (Product="Credit card" OR Product="Mortgage") AND Timely_Response="No" GROUP BY Bank`	0.595s	110.3s	185.4×
USERVISITS	`SELECT SUBSTR(sourceIP, 1, 8), SUM(adRevenue) FROM USERVISITS GROUP BY SUBSTR(sourceIP, 1, 8)`	3.042s	>1000s	>328.7×
USERVISITS, RANKINGS	`SELECT sourceIP, totalRevenue, avgPageRank FROM (SELECT sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM Rankings AS R, UserVisits AS UV WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date('1980-01-01') AND Date('1980-04-01') GROUP BY UV.sourceIP) ORDER BY totalRevenue DESC LIMIT 1`	12.774s	>1000s	>78.3×

Figure 4: Comparison of ObliDB and a baseline where a naïve oblivious database implementation directly ports non-oblivious algorithms to their oblivious counterparts via ORAM. ObliDB outperforms the baseline on all queries. Our aggregate and join queries were run on the Linear storage method because they involve reading most or all rows of the relevant tables.

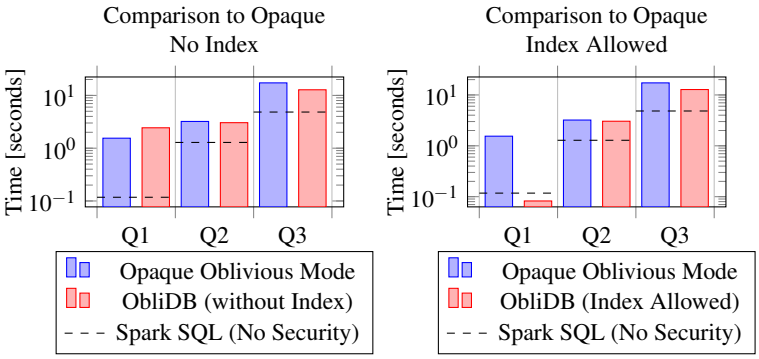

Figure 5: ObliDB outperforms Opaque Oblivious [55] by 1.1-18.8× and never runs more than 2.6× slower than Spark SQL [6] on Queries Q1-Q3 of the Big Data Benchmark [3]. Even without use of an index, ObliDB performs comparably to Opaque Oblivious.

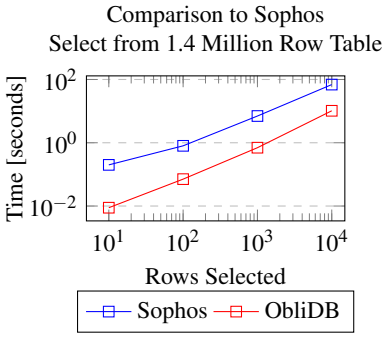

Figure 6: Comparison to Sophos SSE scheme [11]. ObliDB always outperforms Sophos by at least 22.6×. Unlike ObliDB, Sophos leaks access patterns to data.

performs comparably to Opaque in this configuration, slightly worse on query 1 and slightly better on queries 2 and 3. Next, we used the combined storage method. An oblivious index allows ObliDB to outperform Opaque by 18.8× on query 1 since this query scans a small part of a table whereas Opaque and spark SQL, which primarily handle analytic workloads, scan the entire table. Indexes do not provide a speedup on queries 2 and 3 because those queries require scanning most of the input tables. ObliDB is only 2.4× and 2.6× slower than Spark SQL on queries 2 and 3 respectively, putting ObliDB safely in the realm of practical tools for real applications.

We also tested scan-based queries against the Indexed storage method to see how ObliDB would perform on a frequently-updated dataset that is too expensive to maintain in linear storage. These queries performed about **2**× slower than against linear tables. Thus, unlike prior, linear-only systems, ObliDB can perform analytics relatively quickly on "live" tables that are frequently updated through point insertions and deletions.

7.3 Comparison to Sophos

We compare ObliDB's oblivious index to the searchable symmetric encryption (SSE) scheme Sophos [11] in Fig-

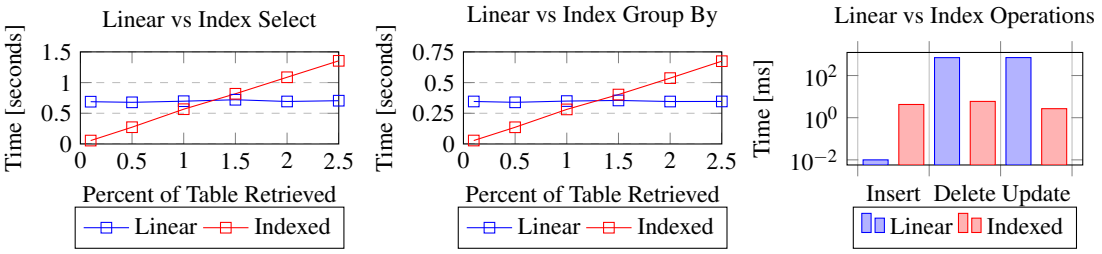

Figure 7: Comparison of Linear and Indexed versions of operators over 100,000 rows of fabricated data. Linear scans do better when more of the data needs to be accessed, but the Indexed storage method performs far better for small queries.

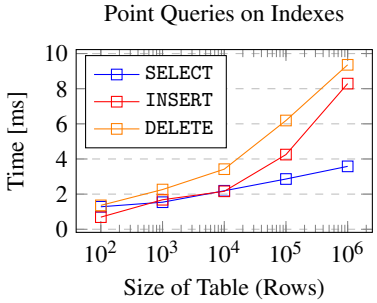

Figure 8: Point queries for tables of various sizes. Query time grows polylogarithmically in table size.

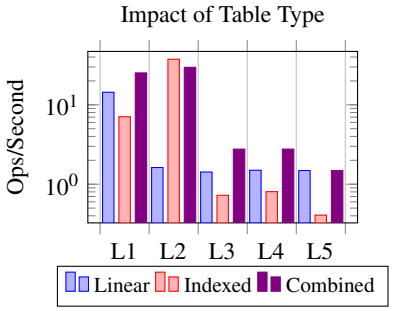

Figure 9: Linear, Indexed, and combined representations of data performing on various workloads over a 100,000 row table. Point reads access 1 record, small reads access 50 records, and large reads access 5% of the table.

Workload	L1	L2	L3	L4	L5
% Point Reads	5	0	50	45	0
% Small Reads	0	90	0	0	0
% Large Reads	5	0	50	45	90
% Insertions	90	9	0	5	5
% Deletions	0	1	0	5	5

ure 6. Sophos does not provide obliviousness guarantees, meaning it leaks access patterns. It does provide a good point of comparison for the performance of our SGX-based oblivious indexes with a non-SGX based, non-oblivious index that still provides some privacy. We compare on simulated data because Sophos only supports exact keyword searches. We compare against numbers reported in the Sophos paper for a 1.4 million row table using a more powerful machine than ours: an Intel Core i7 4790K 4.00GHz CPU with 8 logical cores and 16GB of RAM running on OS X.10. Despite the difference in hardware and the fact the Sophos is multithreaded, ObliDB outperforms Sophos by 22.6-24.6×. We observe that the performance tipping point between Indexed and Linear storage methods in this experiment arrives between 10^4 and 10^5 rows, and ObliDB's performance on larger queries beyond that point would remain constant. ObliDB performs better than Sophos because although it uses ORAM for each memory access, it does not need to execute any costly public-key cryptographic operations, only needing AES to encrypt and decrypt blocks.

7.4 Impact of Table Storage Choices

By providing two storage methods and optimizing queries based on a first pass over data, ObliDB enables meaningful performance improvements for diverse queries. Figure 7 compares our storage methods on `SELECT` (hash algorithm), `GROUP BY` (low-cardinality), `INSERT`, `DELETE`, and `UPDATE` queries. Linear scans perform better when more rows are returned, but smaller queries perform significantly better using an index. Indexed `DELETE` and `UPDATE` queries outperform their linear counterparts, but the fast Linear `INSERT` query outperforms the Indexed `INSERT`. The Linear storage method's performance (outside of constant-time fast insertions) degrades linearly in table size, but point operations on indexes take polylogarithmic time. Figure 8 shows the scaling of various operations. The very gradual increase enables the performance improvements ObliDB enjoys over oblivious analytics systems on queries that admit efficacious use of an index. For comparison, MySQL's point select latency was about .3ms for the 107,000 rows in the CFPB table, but network latency from user to cloud can be tens of milliseconds, rendering the difference insignificant.

Often a combined table representation that maintains both storage methods for the same data proves effective. Although ObliDB pays insertion and deletion costs for both storage methods, it can use the better representation for each query, an important benefit given that many real-world workloads rely heavily on different kinds of reads. Figure 9 shows ObliDB running various workloads with Linear, Indexed, or combined tables. One storage method alone sometimes performs best, but a combined representation often performs better than either alone.

7.5 Impact of Optimizer

Figure 10 demonstrates the effectiveness of ObliDB's choice of `SELECT` algorithms, comparing our various algorithms on queries that retrieve 5% and 95% percent of a 100,000 row table. The "Hash" algorithm performs

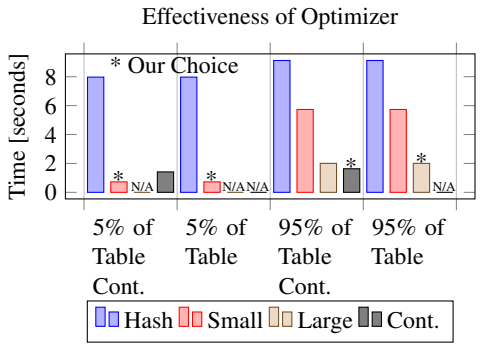

Figure 10: Our optimizer picks the best algorithm for handling `SELECT` queries based on a preliminary scan that determines whether the data to be returned is small, large, or consists of a continuous set of rows in the table.

the best asymptotically, but the figure demonstrates that knowledge gleaned only from ObliDB's intended leakage about the results of a query (whether it is small/large or a continuous set of rows) suffices to pick an algorithm that performs much better in practice. Equally impressive gains appear in the choice of `GROUP BY` algorithm: the high-cardinality aggregation algorithm used for the query on the USERVISITS table (shown in Figure 4) performs 168× better than the low-cardinality algorithm would.

7.6 Padding Mode Overhead

Padding mode additionally hides the sizes of tables, intermediate results, and final outputs. We evaluated this mode by running queries on the CFPB table of 107,000 rows padded to 200,000 rows. Our aggregate query with the Linear storage method had a 4.4× slowdown and a select had a 2.4× slowdown. The larger slowdown for aggregates results from the padding algorithm padding to the maximum supported number of groups for aggregates – in this case, 350,000. We did not evaluate padding mode for indexes because the benefit of the Indexed storage method results from knowledge of the selectivity of a query, the exact information padding hides.

Opaque [55] describes an oblivious padding mode, but does not implement it. To our knowledge, no other comparable system implements an oblivious padding mode, so we are unable to compare to prior work. The results do, however, represent reasonable slowdowns for inflating the size of a table by approximately 2× with padding.

8 Related Work

Cryptographically Protected Databases. Fuller et al [23] summarize prior work on cryptographically protected databases. The well-known CryptDB [36] enables a tradeoff between security and performance, encrypting fields differently according to their security needs. More recently, Arx [35] uses only strong encryption schemes but leverages special data structures to allow search over encrypted data. Other solutions, including Demertzis et al, Sophos, and Diana [11, 12, 17], use searchable encryption.

Although these techniques encrypt data, they do not aim to protect against access pattern leaks. Several attacks [14, 26, 31, 54] show that access pattern leaks compromise these schemes outside their original security models.

Trusted hardware. A number of generic tools provide legacy applications the heightened security available from SGX [7, 10, 25, 45], but these tools do not provide protection against access pattern attacks where applicable. Many works also implement variations of existing systems on SGX [21, 22, 32]. M2R [18] and VC3 [41] provide MapReduce and cloud data analytics functionalities, while HardIDX [22] builds a read-only key-value index in SGX that is is not oblivious. Of these works, Opaque [55] and Cipherbase [4] are the closest to ObliDB because they can optionally support oblivious execution. However, both Opaque and Cipherbase provide oblivious operators only for analytics queries that scan all the data, because they rely on oblivious sorts of an entire input table. To our knowledge, ObliDB is the first oblivious database to support both transactional and analytics workloads.

Several side-channel attacks affect SGX [13, 27, 50, 52], but other work generically closes side channels [38, 42–44]. Outside SGX, some other hardware solutions render programs' memory traces oblivious as well [16, 28, 30].

Data Structures over ORAM. Wang et al. [49] develop several high-level data structures that can be used on top of ORAM. Their tree structures focus on optimizations that reduce overhead for recursive ORAMs, which does not greatly impact our setting (where one-level ORAM is often enough). Roche et al. [39] build a history-independent "HIRB tree" over vORAM, an ORAM with variable-sized blocks. Unlike these systems, ObliDB focuses on optimizing B+ trees and introduces optimizations to reduce the worst-case cost of B+ tree operations and to tune parameters to maximize their performance.

Oblivious Computing ZeroTrace [40] provides oblivious memory primitives based on ORAM over SGX, while ObliVM [29] automatically compiles programs to oblivious representations for use in various cryptographic algorithms. By specializing its data structures and operators to run over ORAM, however, ObliDB greatly outperforms a naïve translation of database algorithms to ORAM.

9 Conclusion

We have presented ObliDB, the first oblivious, enclave-based database that supports both transactional and analytic workloads. ObliDB uses new storage methods (oblivious B+ trees) and operator implementations to support diverse workloads. While obliviousness has a cost, ObliDB performs well in absolute and relative terms: it is competitive with previous oblivious systems for analytics, comes within 2.6× of Spark SQL, and can perform point queries on a 1 million row table with 3–9ms latency.

References

[1] Intel software guard extensions sdk for linux os, developer reference. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf.

[2] Amazon relational database service. https://aws.amazon.com/rds/.

[3] AMPLAB, UNIVERSITY OF CALIFORNIA, B. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[4] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal security with cipherbase. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[5] ARASU, A., AND KAUSHIK, R. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.* (2014), pp. 26–37.

[6] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 1383–1394.

[7] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., GOLTZSCHE, D., EYERS, D. M., KAPITZA, R., PIETZUCH, P. R., AND FETZER, C. SCONE: secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* (2016), pp. 689–703.

[8] AVIRAM, A. F. Interactive b+ tree (c). http://www.amittai.com/prose/bplustree.html.

[9] AZAR, Y., BRODER, A. Z., KARLIN, A. R., AND UPFAL, E. Balanced allocations. *SIAM J. Comput. 29*, 1 (1999), 180–200.

[10] BAUMANN, A., PEINADO, M., AND HUNT, G. C. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[11] BOST, R. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), pp. 1143–1154.

[12] BOST, R., MINAUD, B., AND OHRIMENKO, O. Forward and backward private searchable encryption from constrained cryptographic primitives. *IACR Cryptology ePrint Archive 2017* (2017), 31.

[13] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.* (2017).

[14] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 668–679.

[15] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[16] COSTAN, V., LEBEDEV, I. A., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 857–874.

[17] DEMERTZIS, I., PAPADOPOULOS, S., PAPAPETROU, O., DELIGIANNAKIS, A., AND GAROFALAKIS, M. N. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 185–198.

[18] DINH, T. T. A., SAXENA, P., CHANG, E., OOI, B. C., AND ZHANG, C. M2R: enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 447–462.

[19] ELMASRI, R., AND NAVATHE, S. B. *Fundamentals of Database Systems (6th Edition)*. Pearson, 2010.

[20] ENIGMA. Consumer complaints. https://app.enigma.io/table/us.gov.cfpb.consumer-complaints.

[21] FISCH, B. A., VINAYAGAMURTHY, D., BONEH, D., AND GORBUNOV, S. Iron: Functional encryption using intel sgx. *IACR Cryptology ePrint Archive 2016*.

[22] FUHRY, B., BAHMANI, R., BRASSER, F., HAHN, F., KERSCHBAUM, F., AND SADEGHI, A. Hardidx: Practical and secure index with SGX. In *Data and Applications Security and Privacy XXXI - 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings* (2017), pp. 386–408.

[23] FULLER, B., VARIA, M., YERUKHIMOVICH, A., SHEN, E., HAMLIN, A., GADEPALLY, V., SHAY, R., MITCHELL, J. D., AND CUNNINGHAM, R. K. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), pp. 172–191.

[24] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (1996), 431–473.

[25] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* (2016), pp. 533–549.

[26] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

[27] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR abs/1611.06952* (2016).

[28] LIU, C., HARRIS, A., MAAS, M., HICKS, M. W., TIWARI, M., AND SHI, E. 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, Istanbul, Turkey, March 14-18, 2015* (2015), pp. 87–101.

[29] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 359–376.

[30] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: practical oblivious computation in a secure processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 311–324.

[31] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 644–655.

[32] NAYAK, K., FLETCHER, C. W., REN, L., CHANDRAN, N., LOKAM, S., SHI, E., AND GOYAL, V. Hop: Hardware makes obfuscation practical. In *NDSS*.

[33] NIKOLAENKO, V., WEINSBERG, U., IOANNIDIS, S., JOYE, M., BONEH, D., AND TAFT, N. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 334–348.

[34] OHRIMENKO, O., COSTA, M., FOURNET, C., GKANTSIDIS, C., KOHLWEISS, M., AND SHARMA, D. Observing and preventing leakage in mapreduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1570–1581.

[35] PODDAR, R., BOELTER, T., AND POPA, R. A. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive 2016* (2016), 591.

[36] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: processing queries on an encrypted database. *Commun. ACM 55*, 9 (2012), 103–111.

[37] POPA, R. A., STARK, E., VALDEZ, S., HELFER, J., ZELDOVICH, N., AND BALAKRISHNAN, H. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 157–172.

[38] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 431–446.

[39] ROCHE, D. S., AVIV, A. J., AND CHOI, S. G. A practical oblivious map data structure with secure deletion and history independence. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016), pp. 178–197.

[40] SASY, S., GORBUNOV, S., AND FLETCHER, C. W. Zerotrace : Oblivious memory primitives from intel SGX. *IACR Cryptology ePrint Archive 2017* (2017), 549.

[41] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 38–54.

[42] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*.

[43] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*.

[44] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016* (2016), pp. 317–328.

[45] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXEENA, P. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*.

[46] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 299–310.

[47] THIELMAN, S. Yahoo hack: 1bn accounts compromised by biggest data breach in history, 2016. `https://www.theguardian.com/technology/2016/dec/14/yahoo-hack-security-of-one-billion-accounts-breached`.

[48] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), pp. 299–313.

[49] WANG, X. S., NAYAK, K., LIU, C., CHAN, T. H., SHI, E., STEFANOV, E., AND HUANG, Y. Oblivious data

structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014* (2014), pp. 215–226.

[50] WEICHBRODT, N., KURMUS, A., PIETZUCH, P. R., AND KAPITZA, R. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I* (2016), pp. 440–457.

[51] WU, D. J., ZIMMERMAN, J., PLANUL, J., AND MITCHELL, J. C. Privacy-preserving shortest path computation. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016).

[52] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 640–656.

[53] ZAHUR, S., WANG, X. S., RAYKOVA, M., GASCÓN, A., DOERNER, J., EVANS, D., AND KATZ, J. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016), pp. 218–234.

[54] ZHANG, Y., KATZ, J., AND PAPAMANTHOU, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 707–720.

[55] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), pp. 283–298.