

Membrane: A Cryptographic Access Control System for Data Lakes

Sam Kumar
UCLA

Samyukta Yagati
UC Berkeley

Conor Power
UC Berkeley

David E. Culler
Google

Raluca Ada Popa
UC Berkeley

Abstract

Organizations use data lakes to store and analyze sensitive data. But hackers may compromise data lake storage to bypass access controls and access sensitive data. To address this, we propose Membrane, a system that (1) cryptographically enforces data-dependent access control *views* over a data lake, (2) without restricting the analytical *queries* data scientists can run. We observe that data lakes, unlike DBMSes, disaggregate computation and storage into separate trust domains, making at-rest encryption sufficient to defend against remote attackers targeting data lake storage, even when running analytical queries in plaintext. This leads to a new system design for Membrane that combines encryption at rest with SQL-aware encryption. Using block ciphers, a fast symmetric-key primitive with hardware acceleration in CPUs, we develop a new SQL-aware encryption protocol well-suited to at-rest encryption. Membrane adds overhead only at the *start* of an interactive session due to decrypting views, delaying the first query result by up to $\approx 20\times$; subsequent queries process decrypted data in plaintext, resulting in low amortized overhead.

1 Introduction

Data lakes have emerged as a central paradigm for data analysis. Their key innovation, compared to DBMSes, is to *separate* compute resources (e.g., EC2) from storage resources (e.g., S3) [74]. This has two benefits. First, it allows compute and storage to be scaled independently. Second, it allows data scientists to analyze data using their framework of choice (e.g., Pandas, Spark, etc.). Companies like Microsoft, Google, and Databricks provide data lake platforms [37, 47, 75] based on open file formats (e.g., Parquet [3]) in cloud storage.

Data lakes are increasingly used for sensitive data, like financial data [78] or healthcare data [81]. Thus, each data lake user (e.g., data scientist) must be granted access to only the data that she needs. The state of the art is to use cell-level, *data-dependent* access policies. For example, the pharmaceutical company Eisai demonstrated granting each data scientist access to patient data in only certain US States [58]. This access policy is described by a SQL query/view such as:

```
SELECT * FROM RWE WHERE State IN ("IA", "IL");
```

(1)
We refer to access control (AC) policies based on SQL views as *AC views*. For data lakes, such access control mechanisms are provided directly by data lake platforms [44] and by third-party companies like Immuta [38, 43] and Privacera [39].

Alas, data theft from cloud storage is common [34, 68, 92]. Access control mechanisms help, but are not foolproof. In

2019, for example, an attacker bypassed CapitalOne’s Web Application Firewall (WAF) and gained access to sensitive data in Amazon S3, including 140,000 Social Security numbers and 80,000 bank account numbers [109].

In light of such attacks, it is desirable to encrypt the data lake. This would keep the data protected even if an attacker breaks into the storage. Naïvely using encryption, however, is weak; since all data scientists have the secret key, a *single* compromised data scientist would undermine encryption for the entire data lake. Instead, we want to *cryptographically enforce access control* [28, 54, 69, 93, 105], so each data scientist’s key can decrypt only data she is allowed to access.

How can we design a cryptographic access control system for data lakes? We answer this question with our system Membrane. Membrane encrypts data at the storage servers to remove storage from the Trusted Computing Base (TCB), and ensures that each data scientist can only decrypt and analyze data matching the AC views she is granted. To achieve this, *Membrane is the first system to combine encryption at rest with SQL-aware encryption*. At-rest encryption gives data scientists flexibility to run arbitrary analytics tools like Pandas or SQL, and SQL-aware encryption is used to cryptographically enforce fine-grained, data-dependent access control views.

New system model. One may try using an encrypted file system (EFS) [7, 17, 46, 62, 66, 87, 98, 100, 101], encrypted search system (ESS) [30, 33, 35, 40, 60, 61, 84, 97, 99], or encrypted database (EDB) [13, 28, 30, 32, 41, 53, 63, 64, 83, 84, 86, 88, 94, 95]. However, data lakes have two requirements, flexibility and access control, that render such designs unsuitable.

First, data scientists expect the flexibility to analyze data using *arbitrary* SQL queries or data science/ML frameworks. In contrast, practical EDBs/ESSes limit users to only the small subset of SQL supported cryptographically, and EFSes do not support data-dependent queries (SQL) at all.

Membrane achieves flexibility via at-rest encryption. Data scientists first download encrypted table(s) from storage to their compute nodes, then decrypt the portion of the encrypted table(s) matching AC views that they are granted, and, finally, run analytics on decrypted data *in cleartext* at their compute nodes. Crucially, data analysis is done over *unencrypted* data, so data scientists have the flexibility to issue *arbitrary* SQL queries and use any analytics framework (Pandas, Spark, etc.). Only AC views, usually simpler than analytical queries, are constrained by cryptography. We call this the **Encrypted Data Lake (EDL) model**.

In using at-rest encryption, our insight is that data lakes’ *separation between compute and storage* can enhance the

value of at-rest encryption. To understand how, let us contrast data lakes with DBMSes. In a traditional DBMS, at-rest encryption protects against attackers who only steal a disk drive; in the context of a remote attacker who compromises software, compute and storage are *in the same trust domain* (e.g., the same server). Protecting against such remote attackers when processing data in the cloud requires also removing compute from the TCB, like an EDB; this requires data analytics to compute on encrypted data, limiting EDBs’ flexibility to only queries supported via cryptography [83]. But in a data lake, compute resources are physically and logically distinct from storage resources, as they are often developed/administered by different engineering teams [89]. For example, a remote attacker who gains access to an organization’s storage (e.g., S3 buckets) has not necessarily compromised its compute (e.g., EC2 VMs). Thus, the separation of compute and storage enables Membrane’s at-rest encryption to protect against remote attackers who compromise software. Specifically, Membrane (1) fully removes storage from the TCB, and (2) trusts a data scientist’s compute with *only* data she is granted access to.

In a sense, EFSes/EDBs/ESSes have stronger security than the EDL model, because they hide data from both storage and compute at the cloud provider. So, it may seem natural to adapt EFSes/EDBs/ESSes to the EDL model by weakening their security. Concretely, one could run an EFS/EDB/ESS server in storage and the client in compute, and have data scientists “query” storage for data in their AC views. This fails due to data lakes’ second requirement: access control. Many EDBs/ESSes do not support access control. EFSes, and the few EDBs/ESSes with access control, only allow access policies based on *public* attributes like file paths, not *private* data like the State field (Query 1). Further, this approach requires running a cryptographic protocol in storage, not supported by current cloud storage offerings (e.g., AWS S3).

Membrane delivers access control via a **new SQL-aware encryption protocol** compatible with existing cloud storage offerings. This is possible because Membrane solves a different problem than EDBs. While EDBs compute the *encrypted* result of a SQL query, Membrane’s protocol produces the *plaintext* data of a SQL AC view, using a decryption key for the view. Membrane’s access pattern leakage is limited to which table partitions are fetched from storage. As data lakes often use fast intra-datacenter networks, one can fetch all of a table’s partitions to hide access patterns, if needed (§3).

Designing the protocol. There are several major challenges in designing Membrane’s new SQL-aware protocol.

First, there can be many AC views, so it is untenable to add per-row data for each one. Our insight is that we can add a *single* unit of per-row cryptographic material for a *large set of AC views*. We refer to such sets of AC views as *AC view families* and represent them as query templates. For example, the AC view family `SELECT * FROM RWE WHERE State IN ?x;` includes all 2^{50} AC views for sets of States. Many AC views are often captured by a few AC view families.

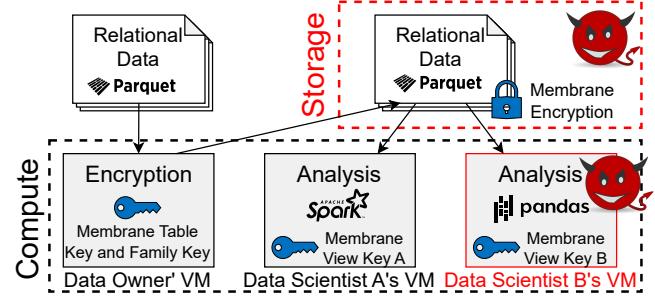


Figure 1: A data lake and how Membrane integrates with it. Membrane components are in blue, and our threat model is in red.

Second, we must map SQL to cryptography. Existing EDBs compute a query plan and have a subprotocol for each step, but this reveals at which step (e.g., which predicate in the `WHERE` clause) each row is filtered out. To avoid leaking partial results, Membrane instead *rewrites* SQL queries into a single monolithic operator called *Membrane-canonical form*.

Third, due to the large scale of data lakes, cryptographic processing must be fast, ideally gigabytes per second. This restricts us to fast symmetric-key tools like the block cipher AES, which, on its own, cannot support SQL and is limited to equality checks. Our insight is to apply an arbitrary function g to cell data *before* applying the block cipher, enabling clauses of the form `g(row) IN ?x`. This may still seem limiting on first glance, but by carefully choosing g , we can actually rewrite *inequalities* (e.g., $<$, \neq), *ANDs*, and *ORs* into this form.

Fourth, because Membrane supports access policies based on *private, encrypted* data, the client cannot easily know which rows of a table they can decrypt. To solve this, we develop *key-hiding tags* that allow a user to identify rows to decrypt *up to 50,000×* faster than naively trying each row.

A 52-core server can decrypt an AC view over 200 GB of in-memory patient data in 2–15 seconds using Membrane. Key-hiding tags provide a speedup of up to 50,000× and are crucial for achieving “big data” speeds. The full process of downloading and decrypting a view takes 30–100 seconds. A limitation of Membrane is that size overheads are up to an order of magnitude, due almost entirely to losing compression when encrypting data. Membrane decrypts views at the *start* of an interactive session; in a PySpark-SQL setup in Databricks, it increases the time to completion for the first analytical query by $\approx 20\times$ compared to non-cryptographic AC views. Membrane is designed so that subsequent analytical queries in the session can use the already-decrypted view, with *no* overhead from Membrane. Thus, its amortized overhead for an interactive session can be small.

2 System Overview

We consider a system model where a *data owner* has a sensitive dataset and wishes to allow *data scientists* to run analytics jobs against the dataset. The data owner specifies what part of the dataset each data scientist can access as an *AC view*.

Membrane enables the data owner to (1) encrypt the dataset, and (2) grant each data scientist cryptographic *view key(s)* that can decrypt only data described by her AC view(s). The data owner can later add rows without re-generating view keys.

The data owner places the encrypted dataset in a data repository, and data scientists run analytics jobs against the data by allocating compute resources distinct from storage. For example, if the data repository is in cloud storage, data scientists may spawn VMs in the same cloud region for analytics jobs. Data scientists analyze data using an *analytics framework* (e.g., Spark-SQL, Pandas, etc.). To process a compute job (e.g., SQL query for Spark-SQL), the framework downloads the relevant data from storage to the compute servers and processes the data there. It can keep data in memory for an interactive session to avoid rereading it on each job.

The overall data lake consists of multiple Membrane deployments, each with a different owner. Each deployment contains some data, and the data owner is a privileged employee (or team) that determines who can access what for that deployment. The data scientists to whom access is granted may be in different teams, or even different organizations altogether.

2.1 Applying Membrane to Data Lakes

Data lakes are varied in applications and deployment models. For example, data lakes are used with unstructured data, such as raw logs, or multimodal data including images and video.

Membrane requires the dataset to be in a structured, relational form. Any unstructured data must first be converted to a structured form (i.e., *cleaned*) before the data owner can apply Membrane. This requirement is fundamental to access control—unstructured data are not in a consistent format, so it is difficult to programmatically enforce access control, cryptographically or otherwise. Moreover, this requirement is consistent with many data lake use cases. Data lake platforms [37, 76, 89] are built on file formats for structured, relational data like Parquet [3] and ORC, and data lake access control offerings [36, 39, 43, 58] target structured (e.g., tabular) data. The established *medallion architecture* [56] involves cleaning data ahead of time, as Membrane requires.

Because Membrane only protects structured data, the process of converting unstructured data to structured data, if applicable to a deployment, must be protected via other means. For example, one can place the unstructured data in a staging area separate from the data lake (e.g., an on-premises cluster). Alternatively, one can use the data lake for this process, but encrypt any unstructured data with a symmetric key held only by the data owner or the party carrying out the conversion.

Strawman #1. To better motivate Membrane’s system model, we compare it to a strawman based on a trusted AC service. The strawman is to protect data in storage with symmetric-key encryption, placing the key at the AC service. The AC service decrypts data and computes AC views for data scientists. It is trusted to see data contents and enforce access control.

Such an AC service is a central point of attack, which Membrane eliminates. This is because an AC service must accept requests from untrusted parties and be highly available, making it much harder to firewall and harden than a data owner, who does not need to host an online service with the secret key. This is the same reason that an EFS design, in which clients store their keys [46], is preferable to having all clients access the file system via a single trusted proxy holding the secret keys. Similar arguments motivate delegable access control [69] and HTTPS, in which Certificate Authorities access root keys rarely to lessen the risk of compromise.

To preserve the benefits of cloud storage, an AC service would need to be engineered to have very high availability and scalability to match cloud storage offerings. The alternative, to integrate the AC service into the storage endpoint, would make it a central point of attack *for the entire data lake*.

2.2 Threat Model and Security Guarantees

Against a malicious adversary \mathcal{A} who has compromised the storage servers and some data scientists (red, Fig. 1), Membrane guarantees that \mathcal{A} cannot see data, except what compromised data scientists are permitted to see by their AC views. We provide a *formal cryptographic security definition and proof* in Appendix B.

In a data lake, large tables (e.g., those gigabytes in size or larger) are usually *partitioned* into multiple Parquet/ORC files, each storing a subset of the table’s rows. If a data scientist fetches only some of a table’s partitions, then \mathcal{A} learns which partitions were fetched. As discussed in §3, one can hide row-level access patterns from \mathcal{A} by having data scientists fetch *all* rows in a table when calling `RevealView`. Even that does not hide *which tables* they access, or *when* they access those tables. Membrane does not hide a table’s schema, the number of rows in a table or partition, or size of each cell. Membrane does not provide anonymity. Membrane does not hide the *positions* of cells accessible to compromised data scientists. To limit leakage via cell positions, one can shuffle rows in a table before encrypting with Membrane.

Membrane is designed to be used with existing techniques for strong integrity guarantees [57, 71, 73]. For example, the data owner may sign updates to files to prevent \mathcal{A} from changing them arbitrarily and sign the entire data repository using a Merkle tree to prevent \mathcal{A} from selectively rolling back files. Such techniques are orthogonal to Membrane’s core contributions and are easy to integrate with Membrane.

2.3 AC View Families

The decryption keys given to data scientists in Fig. 1 grant access to AC views specified as SQL. How can we craft such decryption keys? While functional encryption [20] enables this in theory, it is impractically slow for general functions.

To circumvent this, Membrane slightly relaxes the model: Before Membrane can generate a decryption key for an AC view, the encrypted table must first be augmented with some cryptographic material. However, this must be designed care-

- $\text{EncryptTable}(t) \rightarrow t', k^{\text{tab}}$
 - Input t : table to encrypt
 - Outputs t' , k^{tab} : encrypted table t' and its *table key* k^{tab}
- $\text{AddFamily}(t, k^{\text{tab}}, \text{fam}) \rightarrow t', k^{\text{fam}}$
 - Inputs t, k^{tab} : an encrypted table t and its *table key* k^{tab}
 - Input fam : SQL describing a view family
 - Output t' : new version of t , with fam instantiated
 - Output k^{fam} : *family key* for t' 's instantiation of fam
- $\text{ViewGen}(\text{view}, k^{\text{fam}}) \rightarrow k^{\text{view}}$
 - Input view : SQL describing a view in fam
 - Input k^{fam} : *family key* for an instantiation of fam
 - Output k^{view} : *view key* corresponding to view
- $\text{RevealView}(t, k^{\text{view}}, \text{fil}) \rightarrow t'$
 - Inputs t, fil : encrypted table t and partition filter fil
 - Input k^{view} : *view key* from a family instantiated in t
 - Output: view applied to t 's unencrypted data

Figure 2: Summary of Membrane’s API.

fully; simple approaches requiring adding per-AC-view data to each row are undesirable because there can be many views.

Our insight is that we can add a single unit of per-row cryptographic material for *a large set of AC views*. We refer to such sets of AC views as *AC view families* and refer to the process of adding this material as *instantiating* a view family. We represent AC view families as SQL queries with constants replaced by wildcards. For example, the AC view family `SELECT * FROM customers WHERE Location IN ?x;` includes AC views where $?x$ is replaced by any set of strings (e.g., `WHERE Location IN ("Phoenix", "Mesa")`).

Once an AC view family is instantiated for a table, it is possible to generate a decryption key for any AC view in that AC view family. This is better than adding per-view state to each row because a single AC view family can describe many AC views (e.g., many values for $?x$). This idea, to group views into a small number of patterns (i.e., view families), is inspired by non-cryptographic attribute-based access control [43].

Strawman #2. To better motivate view families, we consider a strawman design that materializes each AC view as a separate Parquet file, and then uses file-level encryption. This has two drawbacks. (1) It requires maintaining *many copies of data* in the data lake. Data lake providers generally avoid this because of the risks of some copies becoming stale and the costs of keeping multiple copies of data up to date [110]. (2) Materializing AC views can have a large storage footprint, particularly if there is a large overlap among AC views.

Instantiating AC view families in Membrane also requires space. Unlike the strawman, Membrane’s extra space scales in the number of *view families*, not *views*. This is a large reduction, as one view family describes many possible views.

2.4 Membrane’s API and Workflow

Fig. 2 summarizes Membrane’s API and describes the EDL model, which we formalize in Appendix B. Upon obtaining a table t (e.g., a set of Parquet files), the data owner (1)

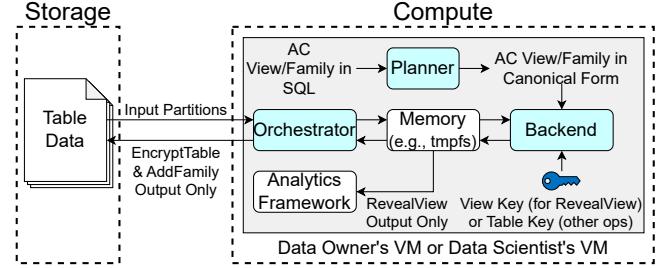


Figure 3: Membrane’s architecture; its components are in blue.

Encrypts t and puts the result in storage; (2) uses `AddFamily` to instantiate their desired AC view families in the encrypted table; and (3) calls `ViewGen` to generate *view keys* for desired AC views, and gives view keys to data scientists to grant them access. The data owner can call `AddFamily` and `ViewGen` at any time to instantiate additional AC view families and grant access to additional AC views. To run analysis jobs, data scientists call `RevealView` to decrypt AC views they are granted.

We envision that data scientists will call `RevealView once per table at the start of an interactive session` to materialize the views locally at the compute servers. Then, they can run compute jobs on these materialized views in plaintext using whatever framework they wish. Essentially, they incur the overhead of `RevealView` once and can then analyze the decrypted view in plaintext indefinitely with no overhead. That said, it may be unavoidable to call `RevealView` again in some cases, so we still designed `RevealView` to be performant.

Membrane does not support modifying data in place; modern data lakes (e.g., Lakehouse systems) often implement logical modifications as physical appends [9, 37].

Access revocation can be achieved, in principle, via *lazy revocation* [62]. The principle is that one cannot make a data scientist “forget” data she was previously granted, but one can hide future rows added after revoking access. Specifically, one can use new family keys when encrypting future rows, and generate fresh view keys from those family keys to grant to users who were not revoked. With this design, old data remain visible to a revoked party, but new data are not.

2.5 System Architecture

Membrane has a *planner*, *backend*, and *orchestrator* (Fig. 3).

2.5.1 Planner (§4)

A critical design decision in Membrane is to use a *single* cryptographic protocol that supports only views/families of a particular form, which we call *Membrane-canonical form*. Membrane has a *planner* that rewrites SQL views/families into Membrane-canonical form, off the critical path. This departs from prior systems, which directly support SQL by composing *multiple* subprotocols for different subexpressions within a SQL query.

To understand why we design Membrane this way, consider EDBs. EDBs like CryptDB have cryptographic subprotocols for each operator (e.g., subexpressions of the `WHERE` clause)

and compose them to execute a query. Consider Query 2:

```
SELECT * FROM t WHERE a = "foo" AND b < 150; (2)
```

An EDB like CryptDB has separate subprotocols to check if a row matches $a = "foo"$ (e.g., deterministic encryption) and if a row matches $b < 150$ (e.g., order-preserving encryption), and would run both at the server to filter out rows. Unfortunately, this does not work for Membrane because a client would learn *which predicates match even for rows outside of the AC view*, leaking information about a and b for those rows.

Membrane avoids this issue because Membrane-canonical form is supported in cryptography as a *single monolith* that does not leak the results of subexpressions.

Choosing Membrane-canonical form is tricky because it must be cryptographically efficient, yet general enough that complex SQL forms can be rewritten to it. We identify the appropriate canonical form to have a `WHERE` clause as a disjunction of predicates $g(\text{row}) \text{ IN } ?x$, where g is an *arbitrary function*. This lets Membrane support **ANDs** and **inequalities** and can be implemented with only fast, symmetric-key cryptography.

2.5.2 Backend (§5)

The backend executes Membrane’s cryptographic protocol on in-memory data. In `EncryptTable/AddFamily/RevealView`, Membrane’s backend can process partitions in parallel on multiple CPUs, producing one output partition for each input partition.

To convey the essence of Membrane’s protocol, we present the following *highly simplified example*. Take the view family `SELECT bname WHERE color IN ?x;` for the table in Fig. 5. In `AddFamily`, (1) for each color (blue, red, green) we sample a *selection key* (k_{blue} , k_{red} , k_{green}), and (2) we encrypt each row’s `bname` with the key corresponding to its color. In `ViewGen`, we map each element of $?x$ to its selection key. For example, for the view `SELECT bname WHERE color IN (red,green);`, the view key is $\{k_{\text{red}}, k_{\text{green}}\}$. In `RevealView`, we use k_{red} to decrypt rows 2 and 4 and k_{green} to decrypt row 3.

Systems like SiRiUS [46] and CryptDB [88, §4] associate encryption keys with filenames and principals, respectively—the core insight in the simplified protocol above is to associate keys with the `color` field as if it were a filename/principal. Membrane’s actual protocol is more complex, as it adds optimizations and levels of indirection to efficiently support multiple view families, multiple `OR` predicates in the `WHERE` clause, etc.

Two additional ideas in Membrane’s protocol are of particular importance. First, instead of having a selection key for each value of a field (e.g., `color`), we have a selection key for each output of an *arbitrary function* g applied to row contents. This allows `WHERE` clauses like $g(\text{row}) \text{ IN } ?x$. Second, whereas filepaths/principals in CryptDB are public, the `color` field in the above example is hidden. Thus, users need a

way in `RevealView` to quickly identify which rows to decrypt and which keys to use. To achieve this, we use ideas from a network middlebox protocol [96] to develop *key-hiding tags*, which allow data scientists to identify rows to decrypt *up to 50,000× faster* than naively trying to decrypt each row.

2.5.3 Orchestrator (§3)

The orchestrator fetches partitions of a table from storage to compute and invokes Membrane’s backend on them. For `EncryptTable/AddFamily`, it writes the output partitions back to storage. For `RevealView`, the output partitions contain decrypted, plaintext data; they are kept at the compute nodes (e.g., in `tmpfs`). A data scientist can then load the decrypted data into an analytics framework on those compute nodes.

2.6 Limitations

Membrane supports only a subset of SQL forms in its AC views (§4.1). In §7, we show that the SQL forms Membrane supports can capture a number of access policies based on realistic use cases. Further, in Membrane, the restrictions on SQL apply only to *AC views*, not *analytical queries*. This is important because analytical queries may be more complex than AC views (see Appendix C).

Differential privacy (DP) [42] is often applied to aggregates. Because Membrane’s views do not support aggregates, they do not support DP. Still, if the table itself contains aggregates, then the data owner can apply DP before calling `EncryptTable`. Similarly, a data scientist who calls `RevealView` and trains an ML model on the result can apply DP to the model.

3 Membrane’s Orchestrator

The orchestrator determines the set of partitions to process (§3) and divides it into *batches*, processed in a streaming fashion. For each batch, it (1) downloads the partitions in that batch from data lake storage to a compute node’s memory, (2) invokes Membrane’s backend to compute the output partitions using multiple CPU cores, and (3) if needed, writes the output partitions back to storage. To overlap computation and I/O, the orchestrator *pipelines* the above stages, processing batches in parallel. Parallel processing of partitions within a batch (§2.5) serves a different role—to use multiple CPU cores.

Choosing which partitions to fetch. For `RevealView`, the orchestrator may need to fetch only a subset of a table’s partitions. For example, an AC view may be fully contained within a subset of a table’s partitions. Or, a data scientist may only wish to analyze part of her AC view. Fetching only some partitions, however, results in *access pattern leakage*—the storage servers learn which rows were or were not fetched. This is not unique to Membrane; most EDBs and ESSes leak which rows/documents their clients query. Sadly, prior research shows that this seemingly innocuous metadata leakage can imply leakage of actual encrypted data [29, 50–52, 80, 107]. Merely partitioning tables with respect to certain columns, as is typical (e.g., so all rows in a partition have the same value for those columns), could leak data via the sizes of partitions.

To reduce such leakage, one can partition tables independently of their contents, and *fetch all partitions* in a table for RevealView. Because data lakes use *local-area* or *intradatacenter* networks (e.g., within a cloud region) where bandwidth is plentiful and cheap, the network cost of fetching all partitions is less significant in data lakes than in EDBs/ESSes.

Still, fetching all partitions may be undesirable due to storage I/O costs. Thus, some users may wish to incur access pattern leakage for better efficiency. The decision may depend on the data semantics and partitioning scheme. For example, consider the NYC Taxi Dataset [11], partitioned by time; access pattern leakage may be acceptable if partitioned at month granularity, but not if partitioned at minute granularity.

Membrane allows data scientists/owners/administrators to choose the best option for each application. Data owners choose how to partition a dataset, and data scientists choose which partitions to fetch via a filter “fil” (Fig. 2), a range of partition IDs in our implementation. If fil is specified, then the orchestrator fetches only matching partitions.

4 Membrane’s Planner and Canonical Form

4.1 Supported SQL Forms

Membrane supports `WHERE` clauses with conjunctions (`AND`) and disjunctions (`OR`) of *predicates*. Each predicate consists of a field, an operation, and a wildcard. The field can be *any deterministic function* applied to the contents of a row. The operation can be $=$, $<$, \leq , $>$, \geq , or \neq for integer types and $=$ or \neq for strings. For example, for a table with column names *a* (string), *b* (integer), and *c* (integer), valid predicates are `LOWER(a) = ?x`, `b ≥ ?x`, or `b² + c ≠ ?x`.

In general, the `SELECT` clause must include fields used in the `WHERE` clause. The reason is that the process of decrypting a row reveals which disjunctive predicates in canonical form are true for that row, leaking information about those fields. Having those fields in the `SELECT` clause makes this explicit.

4.2 Membrane-Canonical Form

Membrane-canonical form is as follows:

```
SELECT columns FROM t WHERE
    g1(row) IN ?x1 OR g2(row) IN ?x2 OR ...; (3)
```

Here, g_1, g_2, \dots are arbitrary functions. For example, suppose that *a* (int), *b* (int), and *c* (string) are columns of the table *t*. Then, the following is in Membrane-canonical form:

```
SELECT a, b, c FROM t WHERE
    b IN ?x1 OR LOWER(c) IN ?x2 OR a + b IN ?x3; (4)
```

Here, `columns` is the list “*a*, *b*, *c*,” g_1 is a function that returns field *b* from a record, g_2 is a function that returns field *c* from a record transformed to lowercase, and g_3 is a function that returns the sum of fields *a* and *b* from a record.

A canonical-form AC view family describes AC views obtained by replacing wildcards $?x_i$ with *sets* of values. For example, the view below belongs to the view family in Query 4:

```
SELECT a, b, c FROM t WHERE
    b = 7 OR b = 8 OR LOWER(c) = "hello"; (5)
```

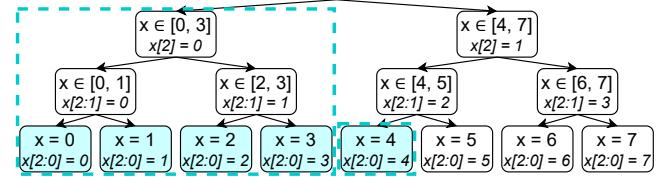


Figure 4: A binary tree covering all 3-bit integers. The range $0 \leq x \leq 4$ is covered by subtrees at $x[2] = 0$ and $x[2:0] = 4$.

Here, x_1 was replaced with $\{7, 8\}$, x_2 was replaced with $\{"hello"\}$, and x_3 was replaced with \emptyset .

4.3 Rewriting AC Views into Canonical Form

To support `AND` we use “secure concatenation”—that is, concatenation that unambiguously delimits the concatenated items. We denote the secure concatenation of *a* and *b* as $a \parallel b$. Our idea is to transform a conjunction of `IN` clauses into a single `IN` clause of secure concatenations. For example, $b \text{ IN } (7) \text{ AND } \text{LOWER}(c) \text{ IN } ("ab")$ becomes $b \parallel \text{LOWER}(c) \text{ IN } (7 \parallel "ab")$. The result grows multiplicatively with the sizes of the input clauses. For example, $b \text{ IN } (7, 8) \text{ AND } \text{LOWER}(c) \text{ IN } ("ab", "cd")$ becomes $b \parallel \text{LOWER}(c) \text{ IN } (7 \parallel "ab", 7 \parallel "cd", 8 \parallel "ab", 8 \parallel "cd")$. We call this as the *combination effect* and measure it in §7.

To support inequalities, we first observe that inequality predicates ($<$, $>$, \leq , \geq , and \neq) can all be transformed into *range* predicates of the form $a \leq x \leq b$. For example, $x < a$ can be transformed into $\ell \leq x \leq a - 1$, where ℓ is the lowest integer. Similarly, $x \neq a$ (equivalently, $x \text{ NOT IN } (a)$) can be transformed into $\ell \leq x \leq a - 1 \text{ OR } a + 1 \leq x \leq h$, where h is the highest integer. Next, we rewrite each range ($a \leq x \leq b$) into a disjunction of `IN` clauses. A naïve approach is to list each value in the range. Instead, our approach, inspired by prior bit tree techniques [69], is to imagine a tree over the domain of the integer type. We represent a range as a list of subtrees *logarithmic in the length of the range*, and represent each subtree as an `IN` clause. Fig. 4 depicts the example $0 \leq x \leq 4$ over a 3-bit integer. The full canonical-form expression for $0 \leq x \leq 4$ is $x[2:0] \text{ IN } (4) \text{ OR } x[2] \text{ IN } (0)$.

Increasing the branching factor makes the tree shallower; this means fewer predicates in the AC view family, but more values per wildcard in the AC view. We discuss this trade-off in §7. We can also support \neq for strings by mapping strings to integers (e.g., with collision-resistant hashing).

5 Membrane’s Backend

Membrane’s cryptographic protocols have the syntax:

- $\text{EncryptTable}(t, k^{\text{tab}}, p) \rightarrow t'$
- $\text{AddFamily}(t, k^{\text{tab}}, \text{fam}, k^{\text{fam}}, p) \rightarrow t'$
- $\text{ViewGen}(\text{view}, k^{\text{fam}}) \rightarrow K^{\text{view}}$
- $\text{RevealView}(t, K^{\text{view}}, p) \rightarrow t'$

These are the operations in Fig. 2, with three minor differences. First, the above protocols operate on a single *partition*, not an entire table; *p* is the partition ID, counting upwards starting at 1 (so $p \neq 0$). Second, while users of Membrane

think of k^{view} as a key, it is actually a *set* of keys. So, in this section, we write k^{view} as K^{view} (capital letter denotes that it is a set). Third, k^{tab} and k^{fam} are *inputs*, not *outputs*, as these are *partition-level* operations, and the same table/family key is used for each partition of the table. For the table-level EncryptTable operation (in Fig. 2), the backend samples k^{tab} uniformly at random, and then uses that k^{tab} for all partitions of the table. For the table-level AddFamily in Fig. 2, the backend accepts k^{tab} (the table key used to encrypt t) as an argument, samples k^{fam} uniformly at random, and then uses those k^{tab} and k^{fam} for all partitions.

5.1 Cryptographic Primitives

A *pseudorandom function* (PRF) is a deterministic function that takes as input a key k and a message x . We denote its application as $\text{PRF}(k, x)$. To a party who does not know k , a PRF’s output, for each x , appears uniformly random. The key k is a λ -bit string; $\lambda = 128$ in our implementation. We typically assume that the message x and output are also λ -bit strings, but sometimes allow x to have arbitrary length.

We denote symmetric-key encryption of message m with key k as $\text{Enc}(k, m)$. The scheme must have two properties. First, it must be CPA-secure [21]: Informally, to any party who does not know k , $\text{Enc}(k, m_1)$ and $\text{Enc}(k, m_2)$, for m_1 and m_2 of equal length, appear to be identically distributed. Second, it must be key-private [1, 14]: Informally, to any party who does not know k_1 and k_2 , $\text{Enc}(k_1, m)$ and $\text{Enc}(k_2, m)$, for any m , appear to be identically distributed. Our PRF and Enc instantiations use the AES block cipher, which has hardware acceleration in commodity x86-64 CPUs via AES-NI.

OTE (“one-time enc”) denotes an encryption scheme optimized for encrypting only a single value (e.g., one-time pad for small messages). As described in §4.3, $a \parallel b$ denotes concatenation of a and b with unambiguous delimitation.

5.2 Protocol Summary

Membrane’s backend consists of three layers: *projection*, *selection*, and *tagging*. AddFamily adds a column for each layer in order; RevealView works in the opposite order.

The *projection layer* handles the `SELECT` clause. For AddFamily, this layer adds a *projection column* to the table and computes a *projection key*, k_r^{proj} , for each row (index r). In RevealView, the projection key for a row is used with the projection column to decrypt the `SELECTed` fields for that row.

The *selection layer* handles the `WHERE` clause. For AddFamily, this layer adds a *selection column* to the table, containing an encryption of k_r^{proj} for each row (index r). For each row, it computes a set K_r^{sel} of *selection keys*. K^{view} contains one key per wildcard value $?x_j$ in the view. Crucially, they are computed such that the rows matching the view are exactly the rows for which $K^{\text{view}} \cap K_r^{\text{sel}} \neq \emptyset$. In RevealView, the user decrypts the selection column for each matching row using a key in $K^{\text{view}} \cap K_r^{\text{sel}}$, to get k_r^{proj} . k_r^{proj} is used to decrypt the projection column to get `SELECTed` fields in matching rows.

K^{view} may contain many keys. The *tagging layer* adds a

tagging column, used in RevealView to quickly determine which key(s) in K^{view} are in K_r^{sel} . The column contains, in each row, a tag computed from each key in K_r^{sel} . Checking if a key in K^{view} matches a tag can be far more efficient than naively trying to decrypt the row using each key in K^{view} .

We provide a **full protocol description** in Appendix A. Below, we explain our protocol using the AC view family in Query 6 and the table in Fig. 5 as a running example.

```
SELECT bname, color WHERE
    bname IN ?x1 OR color IN ?x2; (6)
```

r	bid ($c = 1$)	bname ($c = 2$)	color ($c = 3$)
1	101	Interlake	blue
2	102	Interlake	red
3	103	Clipper	green
4	104	Marine	red

Figure 5: Relation of boats’ IDs, names, and colors [91].

5.3 The EncryptTable Operation

EncryptTable encrypts each cell of a table with a separate key, called a *cell key*. Cell keys are a layer of indirection—AddFamily encrypts the cell keys according to the view family, and RevealView first decrypts the cell keys for cells matching an AC view and then uses the cell keys to decrypt cell data.

The party running EncryptTable (i.e., the data owner) need *not* remember the cell keys. The data owner only stores k (λ bits) for each table; EncryptTable uses a PRF to derive cell keys from the table key k on the fly. Concretely, each cell in a partition of t is identified by its row index r within the partition and its column index c . For each row in partition p , we derive a *row key*¹ from k as $k_r := \text{PRF}(k, p \parallel r)$. For each cell in row r , we derive a *cell key* from k_r as $k_{r,c} := \text{PRF}(k_r, c)$. Then, we encrypt each cell using its cell key. The cell keys are not used to encrypt anything else (Membrane does not allow edits in place), so we use OTE. See Fig. 6.

r	bid ($c = 1$)	bname ($c = 2$)	color ($c = 3$)
1	OTE($k_{1,1}$, 101)	OTE($k_{1,2}$, Interlake)	OTE($k_{1,3}$, blue)
2	OTE($k_{2,1}$, 102)	OTE($k_{2,2}$, Interlake)	OTE($k_{2,3}$, red)
3	OTE($k_{3,1}$, 103)	OTE($k_{3,2}$, Clipper)	OTE($k_{3,3}$, green)
4	OTE($k_{4,1}$, 104)	OTE($k_{4,2}$, Marine)	OTE($k_{4,3}$, red)

Figure 6: Result of EncryptTable applied to Fig. 5.

An alternative design is to not have row keys, and instead derive cell keys directly from the table key as $k_{r,c} := \text{PRF}(k, p \parallel r \parallel c)$. Row keys, however, enable a space-saving optimization in the projection layer, as we shall see next.

5.4 Projection Layer

In AddFamily, Membrane samples a random λ -bit projection key k_r^{proj} for each row (in the general case—see optimizations below). The projection column contains the cell keys for the columns in the `SELECT` clause (i.e., in the

¹Each partition has its own space of keys. To make this explicit, we could have denoted row keys as $k_{p,r}$ instead of k_r , denoted cell keys as $k_{p,r,c}$ instead of $k_{r,c}$, and similarly carried an additional p subscript throughout.

projection_fields list), encrypted using k_r^{proj} . More formally, if c_1, \dots, c_p are the indices of SELECTED columns, then the projection column contains $\text{Enc}(k_r^{\text{proj}}, k_{r,c_1} || \dots || k_{r,c_p})$. In RevealView, a user with k_r^{proj} can decrypt the cell keys in the projection column and use those cell keys to decrypt the fields in row r corresponding to the SELECTED columns.

A user whose view does not include a row r may, in RevealView, arrive at the projection layer with the wrong value for k_r^{proj} . Therefore, AddFamily also includes $\text{Enc}(k_r^{\text{proj}}, 0)$ in the projection column. This lets RevealView identify if k_r^{proj} is incorrect and omit row r from the output if so. See Fig. 7.

r	projection key	projection column (p. col.)
1	k_1^{proj} samp. rand.	$\text{Enc}(k_1^{\text{proj}}, k_{1,2} k_{1,3}) \text{Enc}(k_1^{\text{proj}}, 0)$
2	k_2^{proj} samp. rand.	$\text{Enc}(k_2^{\text{proj}}, k_{2,2} k_{2,3}) \text{Enc}(k_2^{\text{proj}}, 0)$
3	k_3^{proj} samp. rand.	$\text{Enc}(k_3^{\text{proj}}, k_{3,2} k_{3,3}) \text{Enc}(k_3^{\text{proj}}, 0)$
4	k_4^{proj} samp. rand.	$\text{Enc}(k_4^{\text{proj}}, k_{4,2} k_{4,3}) \text{Enc}(k_4^{\text{proj}}, 0)$

Figure 7: Projection layer for AC view family in Query 6.

There are two space-saving optimizations. If only one column is SELECTED, say c_1 , then k_r^{proj} is set to the cell key for that column, k_{r,c_1} . If all columns are SELECTED (SELECT *), then k_r^{proj} is set to the row key, k_r . In these cases, encrypted cell keys are omitted from the projection column, saving space, and $\text{Enc}(k_r^{\text{proj}}, 0)$ is replaced with $\text{PRF}(k_r^{\text{proj}}, 0)$.

5.5 Selection Layer

In a Membrane-canonical view family, the WHERE clause has n predicates ORed together. The j th predicate has the form $g_j(\text{row}) \text{ IN } ?x_j$ (see Query 3). For each row (index r) and predicate (index j), we choose a selection key $k_{r,j}^{\text{sel}}$. In AddFamily, we encrypt each row’s projection key once per predicate as $\text{Enc}(\text{PRF}(k_{r,j}^{\text{sel}}, 0), k_r^{\text{proj}})$ and put the n ciphertexts in the selection column. For example, see Fig. 8. The selection keys $k_{r,j}^{\text{sel}}$ are chosen such that, if row r satisfies predicate j for a view, then $k_{r,j}^{\text{sel}} \in K^{\text{view}}$. Thus, in RevealView, we can decrypt the ciphertext for that predicate and get k_r^{proj} .

r	selection column (s. col.)
1	$\text{Enc}(\text{PRF}(k_{1,1}^{\text{sel}}, 0), k_1^{\text{proj}}) \text{Enc}(\text{PRF}(k_{1,2}^{\text{sel}}, 0), k_1^{\text{proj}})$
2	$\text{Enc}(\text{PRF}(k_{2,1}^{\text{sel}}, 0), k_2^{\text{proj}}) \text{Enc}(\text{PRF}(k_{2,2}^{\text{sel}}, 0), k_2^{\text{proj}})$
3	$\text{Enc}(\text{PRF}(k_{3,1}^{\text{sel}}, 0), k_3^{\text{proj}}) \text{Enc}(\text{PRF}(k_{3,2}^{\text{sel}}, 0), k_3^{\text{proj}})$
4	$\text{Enc}(\text{PRF}(k_{4,1}^{\text{sel}}, 0), k_4^{\text{proj}}) \text{Enc}(\text{PRF}(k_{4,2}^{\text{sel}}, 0), k_4^{\text{proj}})$

Figure 8: Selection layer for AC view family in Query 6.

How are selection keys and view keys generated? For each AC view family, Membrane uses a random λ -bit view family key k^{fam} . For each predicate in the view family (index j), we derive a predicate key as $k_j^{\text{pred}} := \text{PRF}(k^{\text{fam}}, j)$. In AddFamily, selection keys are derived from the predicate key as $k_{r,j}^{\text{sel}} := \text{PRF}(k_j^{\text{pred}}, g_j(\text{row}))$. (See §4.2 for an explanation of g_j .) ViewGen generates a view key K^{view} as follows. A view assigns a list of wildcard values to each predicate in the view family; for each wildcard value x assigned to predicate

j , the view key K^{view} contains the key $k_{j,x}^{\text{view}} := \text{PRF}(k_j^{\text{pred}}, x)$. Observe that if a wildcard value x for predicate j equals $g_j(\text{row})$ for a row, then $k_{r,j}^{\text{sel}} = k_{j,x}^{\text{view}}$, allowing the user to decrypt the j th ciphertext in the selection column for row r and obtain k_r^{proj} , as desired. This works because selection keys and view keys are both derived from k^{fam} . See Fig. 9.

r	selection keys (for Fig. 8)
1	$k_{1,1}^{\text{sel}} := \text{PRF}(k_1^{\text{pred}}, \text{Interlake}), k_{1,2}^{\text{sel}} := \text{PRF}(k_2^{\text{pred}}, \text{blue})$
2	$k_{2,1}^{\text{sel}} := \text{PRF}(k_1^{\text{pred}}, \text{Interlake}), k_{2,2}^{\text{sel}} := \text{PRF}(k_2^{\text{pred}}, \text{red})$
3	$k_{3,1}^{\text{sel}} := \text{PRF}(k_1^{\text{pred}}, \text{Clipper}), k_{3,2}^{\text{sel}} := \text{PRF}(k_2^{\text{pred}}, \text{green})$
4	$k_{4,1}^{\text{sel}} := \text{PRF}(k_1^{\text{pred}}, \text{Marine}), k_{4,2}^{\text{sel}} := \text{PRF}(k_2^{\text{pred}}, \text{red})$

Figure 9: Selection keys for Fig. 8. Note that $k_1^{\text{pred}} := \text{PRF}(k^{\text{fam}}, 1)$ and $k_2^{\text{pred}} := \text{PRF}(k^{\text{fam}}, 2)$.

Given that the ciphertexts in the selection column are computed using a key derived from table data, key-privacy of the encryption scheme is crucial for Membrane’s security.

It may seem tempting to not have predicate keys, and instead derive selection keys and view key members directly from the view family key as $k_{r,j}^{\text{sel}} := \text{PRF}(k^{\text{fam}}, g_j(\text{row}))$ and $k_{j,x}^{\text{view}} := \text{PRF}(k^{\text{fam}}, x)$. This is insecure; it would allow a view key for bname = “blue” to decrypt row 1, for example.

An alternate design is to encrypt projection keys with $k_{r,j}^{\text{sel}}$, as $\text{Enc}(k_{r,j}^{\text{sel}}, k_r^{\text{proj}})$. We prefer encrypting with $\text{PRF}(k_{r,j}^{\text{sel}}, 0)$, as it enables key-hiding tags (§5.6) to use Enc as a black box.

5.6 Tagging Layer

So far the user, in RevealView, must try decrypting each row with each key in K^{view} . This can be slow. The tagging layer addresses this with a *tagging column* containing, for each row, a *tag* for each selection key. These tags let the user identify, with high probability, *without any cryptographic operations for those rows*, rows that they cannot decrypt.

The challenge is that tags may leak information. For example, computing the tag by cryptographically hashing $k_{r,j}^{\text{sel}}$ provides the desired functionality, but is insecure—if two rows have the same tag, a user can deduce that they have the same selection key, and therefore the same predicate value.

To solve this, we develop *key-hiding tags*. We identify BlindBox Detect [96], a protocol for network middleboxes, as a starting point. The idea is to generate a *different* tag each time a key is used, by applying a PRF to the key and a *count* of how many previous rows use that key in the same predicate. Using this in Membrane requires a *stateful* scan of a table, as Membrane must maintain a counter for each selection key in AddFamily and for each key in K^{view} in RevealView. Alas, this complicates parallel execution, as counters for a row are not known until all previous rows are processed.

Our solution is to generate tags using a *different key* for each partition. This way, partitions can be processed in parallel—although counters in different partitions may collide, the tags will appear independent. Specifically, we generate a *tagging key* as $\tau_{k_{r,j}^{\text{sel}}} := \text{PRF}(k_{r,j}^{\text{sel}}, p)$ for each of the row’s

r	$K^{\text{view}} \cap K_r^{\text{sel}}$	decryption flow	count $_k$ after processing row
1	$\{k_{1,\text{Interlake}}^{\text{view}}\}$	$k_{1,\text{Interlake}}^{\text{view}} \xrightarrow{\text{s. col.}} k_1^{\text{proj}} \xrightarrow{\text{p. col.}} k_{1,2}, k_{1,3} \xrightarrow{\text{cells}} \text{Interlake, blue}$	$\text{count}_{k_{1,\text{Interlake}}^{\text{view}}} = 1, \text{count}_{k_{2,\text{red}}^{\text{view}}} = 0$
2	$\{k_{1,\text{Interlake}}^{\text{view}}, k_{2,\text{red}}^{\text{view}}\}$	$k_{1,\text{Interlake}}^{\text{view}} \text{ or } k_{2,\text{red}}^{\text{view}} \xrightarrow{\text{s. col.}} k_2^{\text{proj}} \xrightarrow{\text{p. col.}} k_{2,2}, k_{2,3} \xrightarrow{\text{cells}} \text{Interlake, red}$	$\text{count}_{k_{1,\text{Interlake}}^{\text{view}}} = 2, \text{count}_{k_{2,\text{red}}^{\text{view}}} = 1$
3	\emptyset	Cannot decrypt s. col.; count $_k$ values and NETs are unchanged	$\text{count}_{k_{1,\text{Interlake}}^{\text{view}}} = 2, \text{count}_{k_{2,\text{red}}^{\text{view}}} = 1$
4	$\{k_{2,\text{red}}^{\text{view}}\}$	$k_{2,\text{red}}^{\text{view}} \xrightarrow{\text{s. col.}} k_4^{\text{proj}} \xrightarrow{\text{p. col.}} k_{4,2}, k_{4,3} \xrightarrow{\text{cells}} \text{Marine, red}$	$\text{count}_{k_{1,\text{Interlake}}^{\text{view}}} = 2, \text{count}_{k_{2,\text{red}}^{\text{view}}} = 2$

Figure 10: RevealView for running example with wildcards $x_1 = \{\text{Interlake}\}$ and $x_2 = \{\text{red}\}$. ViewGen outputs $K^{\text{view}} = \{k_{1,\text{Interlake}}^{\text{view}}, k_{2,\text{red}}^{\text{view}}\}$, where $k_{1,\text{Interlake}}^{\text{view}} = \text{PRF}(\text{PRF}(k^{\text{fam}}, 1), \text{Interlake})$ and $k_{2,\text{red}}^{\text{view}} = \text{PRF}(\text{PRF}(k^{\text{fam}}, 2), \text{red})$. Both count $_k$ values start at 0.

selection keys. For each selection key $k_{r,j}^{\text{sel}}$ the corresponding tag is $\text{PRF}(\tau_{k_{r,j}^{\text{sel}}}, \text{count}_{k_{r,j}^{\text{sel}}})$, where count $_k$ is the number of previous rows for which k is a selection key. In RevealView, the user calculates the *next expected tag (NET)* for each key $k_{j,x}^{\text{view}}$ as $\text{PRF}(\tau_{k_{j,x}^{\text{view}}}, \text{count}_{k_{j,x}^{\text{view}}})$ and maintains the NETs in a data structure with efficient lookup, like a hash set. For each row, the user checks if a tag in the tagging column matches a NET in the data structure. If there is a match, then the user decrypts the row; otherwise, the user skips the row without performing cryptographic operations. Then, for *each* key k in K^{view} that can decrypt that row, the user increments count $_k$, recalculates k 's NET, and updates the data structure.

To save space, we truncate tags, as BlindBox does with RS. This allows false positives—truncated tags may match where full tags do not—but false positives will be caught at the projection layer, when checking $\text{Enc}(k^{\text{proj}}, 0)$.

An alternative design to key-hiding tags could be to adapt an ESS' index structure to Membrane's setting.

Fig. 10 shows RevealView, including the tagging layer.

6 Implementation

We defined Membrane-canonical form in Protobuf [48]. While canonical views, in principle, can have arbitrary predicate functions g_j , our implementation supports selecting a field, bits of a field, and concatenations of those results. This is enough to support equalities and inequalities on fields, but not `UPPER` or `LOWER`. Not-equal (\neq) queries on strings are supported by first hashing strings to integers with SHA-256.

6.1 Membrane's Backend

We wrote the backend in C++, using AES-128 block cipher as a PRF (more details in Appendix B.3). For efficiency, our implementation applies it on batches of input blocks, accelerated with AES-NI instructions. We instantiate Enc using CPA-secure counter-mode encryption; to optimize storage costs, we choose nonces deterministically based on the cell position, while ensuring they are far enough apart to avoid overlap. For OTE, we use a one-time pad for short inputs, and counter-mode encryption with a zero nonce for longer inputs.

Membrane's backend provides a C++ API that can perform Encrypt, AddFamily, ViewGen, and RevealView operations based on the canonical view for a view family or view. Given a batch of partitions, provided as files (e.g., in an in-memory file system), it can parallelize Encrypt, AddFamily, and RevealView operations by processing partitions on different CPU cores. We use Apache Arrow [5] to manipulate relations in

Membrane's backend because of its ability directly interface a wide range of data analytics tools, including Spark [10, 106], Pandas [82], and DuckDB [90], and various relational file formats like Parquet [3], ORC [6], and CSV.

We also implemented an optimization to AddFamily that we call the *selection cache*. In each row, for each predicate, Membrane's backend must use the selection key $k_{r,j}^{\text{sel}}$ to compute the encryption key $\text{PRF}(k_{r,j}^{\text{sel}}, 0)$ in the selection column and a tagging key $\tau_{k_{r,j}^{\text{sel}}}$ in the tagging column. Our insight is that the same value often appears in the same column in multiple rows—for example, a State column may have multiple rows containing CA. Depending on the predicate (e.g., a predicate like `State =? x`), such repetitions may cause multiple rows to use the same selection key $k_{r,j}^{\text{sel}}$ for a predicate. The selection cache is a mapping from $k_{r,j}^{\text{sel}}$ to the prepared AES key schedules for the selection column encryption key and tagging key, to save the work of re-computing these keys when the selection key appears more than once. The selection cache for a predicate has limited capacity; we use an LRU eviction policy, re-computing the keys on a miss and using the precomputed keys on a hit. This makes the cache effective when the selection key repeats in nearby rows, without consuming excessive memory for predicates for which the selection key does not repeat (or repeats rarely).

6.2 Membrane's Orchestrator

We wrote the orchestrator in Python. It uses cloud-provided tools (e.g., `azcopy`) to efficiently transfer data between memory and storage (Azure Data Lake Storage). It invokes `azcopy` and Membrane's backend by spawning them as separate processes. Membrane's backend operates on files; batches are passed between the processes via an *in-memory* file system (e.g., `/dev/shm`). This lets the orchestrator use cloud-provided tools (e.g., `azcopy`) to efficiently transfer data between memory and storage.

6.3 Membrane's Planner

We implemented Membrane's planner in Rust. It transforms SQL statements into Membrane-canonical form via a series of AST transformations, which we describe below.

Immediately after parsing a SQL statement, the AST has `ANDs`, `ORs`, and `NOTs` as internal nodes, and inequalities, equalities, and `INs` as leaf nodes. First, the planner applies De Morgan's Laws to push `NOTs` down into the leaves. Second, it transforms all leaves into ranges using the above rules. Third, as an optimization, it combines multiple ranges on the same

field into as few ranges as possible. Fourth, it converts ranges into [INs](#) using the above rules. Fifth, it applies the distributed law to transform the AST into disjunctive normal form (DNF). Sixth, it eliminates [AND](#) internal nodes using the above rules. DNF is necessary because our technique for [ANDs](#) only works on [ANDs](#) where all children are leaves.

The planner optimizes the AST to produce an output with fewer predicates. The key optimization is to *consolidate* siblings operating on the same field. For example, while a SQL statement may specify `x = "hello" OR x = "world"`, the planner will consolidate these two clauses into a single predicate `x` with two wildcard values, "hello" and "world", instead of naively producing two predicates each with a single wildcard value. Similarly, siblings in the tree that are range predicates on the same field can be consolidated into a single range predicate with multiple range values. Thus, conjunctions of inequalities on the *same* field are simplified by the optimizer into a single array of bit selection predicates. Conjunctions of inequalities on *different* fields cannot be so optimized; each such inequality is converted to a disjunction of bit-selection, and the conjunction of disjunctions is “multiplied out” when transforming the AST into DNF. This results in a canonical form with many predicates.

7 Evaluation

We use three datasets: (1) a synthetic medical dataset generated using Synthea [55], (2) New York City yellow taxi trip records [11], and (3) LHBench, a TPC-DS-based dataset for benchmarking data lakes [59]. We use two versions of each—a small version for testing single-core performance (max table size \approx 2 GiB, uncompressed), and a large version to test scalability. In the large version, RWE (“real-world medical evidence” table created using Synthea²) was \approx 200 GiB uncompressed, yellow (table from NYC dataset) was \approx 250 GiB uncompressed, and store_sales (table from LHBench) was \approx 1 TiB uncompressed. In the large version, medications and conditions tables (from Synthea) were only tens of GiB. **Views based on realistic use cases.** The first AC view, **rwe_state**, is the Eisai example [58] from §1, granting access to rows from RWE for certain US States. The Eisai demo also discusses the importance of hiding entire columns; this inspires **rwe_obs_state**, which is the same as **rwe_state** except that it only `SELECTs` 9 columns. SMCQL [13] describes an analytical query counting patients prescribed aspirin and then diagnosed with heart disease. This inspires our next two AC views: **medication** grants access to data for certain medication codes (aspirin) in Synthea’s medications table, and **diagnosis** grants access to diagnoses of certain condition codes (heart disease) in Synthea’s conditions table. Location privacy is an important concern for datasets like NYC Taxi Cab [70],

²Synthea does not directly output RWE. To produce an RWE table similar to Eisai’s demo, we join the Patient, Observation, and Encounter tables. All Synthea tables were produced by running Synthea separately for each US State, with sizes proportional to their populations, and combining the data.

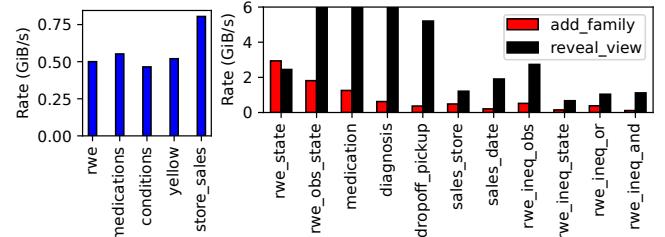


Figure 11: Throughput of Membrane’s backend on in-memory data.

inspiring our next AC view: **dropoff_pickup** grants access to taxi rides for particular combinations of dropoff and pickup zones, using Membrane’s support for [AND](#). LHBench includes sales and customer data. Like the Eisai example, **sales_store** grants access to data in **store_sales** for a particular set of stores. As historical data is often restricted, **sales_date** grants access to data in **store_sales** for a particular time range.

Views to stress Membrane. These AC views all use RWE; we vary only the SQL. **rwe_ineq_obs** grants access to patient observations in a time range. **rwe_ineq_state** does an inequality check on *strings*, requiring hashing strings to integers and forming a tall tree over 256-bit integers. **rwe_ineq_or** is an [OR](#) of the inequality in **rwe_ineq_obs** and an inequality checking that a patient’s death date is not NULL. **rwe_ineq_and** is like **rwe_ineq_or**, but is an [AND](#) of the inequalities instead of an [OR](#). This requires “multiplying out” the conjunction into DNF (§4.3), creating many predicates.

SQL for these views is in Table 1.

7.1 Membrane’s Cryptographic Protocol

We measure Membrane’s backend’s performance on a single core using the small version of the datasets. To measure backend performance, we divide the time to process each table in memory (excluding reading/writing the input/output) by the *uncompressed* plaintext size (Fig. 11). **Membrane’s backend runs at hundreds of megabytes to gigabytes per second on a single core**, showing that our design based on hardware-accelerated, symmetric-key cryptography is performant.

EncryptTable is fastest for **store_sales** because all of its cells are at most 16 bytes, so OTE can use the fast one-time pad for all cells. AddFamily is faster for many-column tables (e.g., RWE) because AddFamily only computes on columns in the `WHERE` clause (a small fraction of a many-column table).

RevealView is much faster for AC views that match fewer rows (i.e., have low selectivity) because, with key-hiding tags (§5.6), it performs no cryptography for non-matching rows. Fig. 12 varies the State in the `WHERE` clause for **rwe_state**, showing that RevealView performance is linear in selectivity.

AC view families whose canonical form has *many predicates* are generally slower to process. This affects inequalities, which are rewritten to many predicates (§4.3), and particularly conjunctions of ranges, which must be “multiplied out” to DNF. For example, **rwe_ineq_and** has low AddFamily throughput, and similar RevealView throughput as **rwe_ineq_or**, despite matching fewer rows.

Workload	View Family SQL	View Wildcard Values	s_1 (%)	s_2 (%)
rwe_state	<code>SELECT * FROM rwe WHERE PATIENT_STATE = ?x;</code>	[Alaska, California, Hawaii, Oregon, Washington]	14.16	15.17
medication	<code>SELECT * FROM medications WHERE CODE = ?x;</code>	[212033, 243670, 2563431]	0.34	0.38
diagnosis	<code>SELECT * FROM conditions WHERE CODE = ?x;</code>	414545008	0.32	0.32
rwe_obs_state	<code>SELECT OBSERVATION_DATE, OBSERVATION_CATEGORY, OBSERVATION_CODE, OBSERVATION_DESCRIPTION, OBSERVATION_VALUE, OBSERVATION_UNITS, OBSERVATION_TYPE, PATIENT_STATE FROM rwe WHERE PATIENT_STATE = ?x;</code>	[Alaska, California, Hawaii, Oregon, Washington]	14.16	15.17
dropoff_pickup	<code>SELECT * FROM yellow WHERE doLocationId = ?x AND puLocationId = ?y;</code>	[(236, 236), (236, 237), (237, 236), (237, 237)]	2.12	0.40
sales_store	<code>SELECT * FROM store_sales WHERE ss_store_sk = ?x;</code>	7	15.80	0.14
sales_date	<code>SELECT * FROM store_sales WHERE ss_sold_date_sk > ?x AND ss_sold_date_sk < ?y;</code>	(2452411, 2452642)	14.89	14.94
rwe_ineq_obs	<code>SELECT * FROM rwe WHERE OBSERVATION_DATE ≥ ?x;</code>	"2022-01-01 00:00:00+00:00"	14.64	14.17
rwe_ineq_state	<code>SELECT * FROM rwe WHERE PATIENT_STATE ≠ ?x;</code>	California	90.51	89.06
rwe_ineq_or	<code>SELECT * FROM rwe WHERE PATIENT_DEATHDATE ≠ ?x OR OBSERVATION_DATE ≥ ?y;</code>	(NULL, "2022-01-01T00:00:00+00:00")	43.05	47.30
rwe_ineq_and	<code>SELECT * FROM rwe WHERE PATIENT_DEATHDATE ≠ ?x AND OBSERVATION_DATE ≥ ?y;</code>	(NULL, "2022-01-01T00:00:00+00:00")	0.43	0.26

Table 1: Workloads to evaluate Membrane. Selectivity for the “small” version of the datasets is s_1 , and selectivity for the “large” version is s_2 .

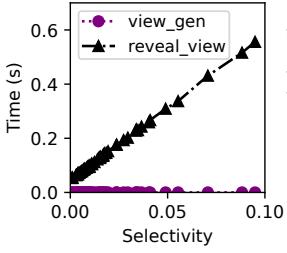


Figure 12: Selectivity.

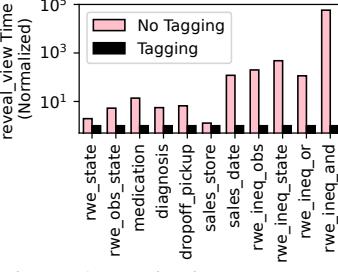


Figure 13: Tagging in RevealView.

ViewGen (not graphed) is most expensive for many-predicate AC views. For a branching factor of 256 (our default) for inequality trees (§4.3), its overheads are modest: ≈ 84 ms runtime and ≈ 438 MB memory for **rwe_ineq_and**, and ≈ 30 ms runtime and ≈ 31 MB memory for **rwe_ineq_or**.

rwe_obs_state `SELECTs` only some columns. This speeds up **RevealView**, as **RevealView** decrypts `SELECTED` columns only. It slows **AddFamily** because it precludes the optimization that omits encrypted cell keys and takes $k_r^{\text{proj}} = k_r$ (§5.4).

rwe_ineq_state involves string inequality. This slows **AddFamily** because it computes a SHA-256 hash per row.

7.2 Key-Hiding Tags and Selection Cache

As shown in Fig. 13, **key-hiding tags speed up RevealView by $1.3\times$ to over $50,000\times$** . For inequality-based views in particular, key-hiding tags are essential to achieving “big data” speeds. This is because the planner rewrites inequalities into disjunctions of many predicates; for a branching factor of 256, K^{view} can contain *hundreds* of keys per predicate. With key-hiding tags, the client uses a hash table to quickly find the key to use in K^{view} ; without them the client must try decrypting each row with *each key* in K^{view} , which is slow when K^{view} is large. The gains are especially significant for view containing inequalities. **rwe_ineq_and** shows an extreme performance gain for two reasons. First, conjunctions of inequalities are “multiplied out” to DNF, and K^{view} also grows multiplicatively due to the *combination effect* (§4.3). Second, its selectivity is

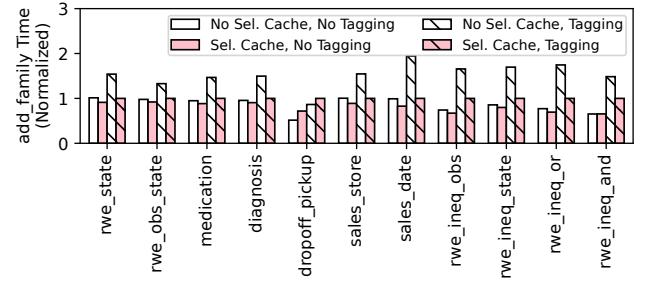


Figure 14: Feature impact (AddFamily).

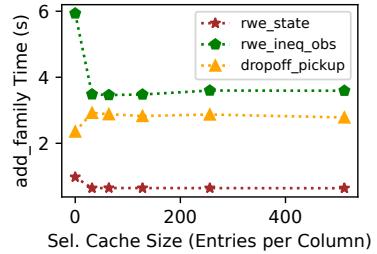
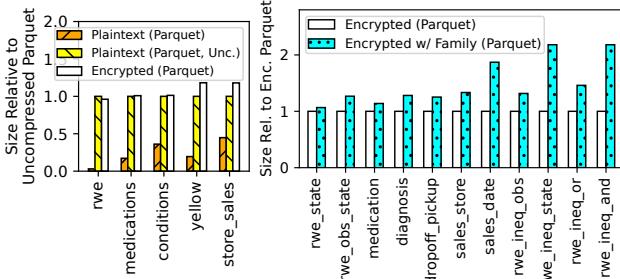


Figure 15: Selection cache size.

< 1%, so **RevealView**, with key-hiding tags, handles > 99% of rows without any cryptographic operations.

Fig. 14 shows how tagging and the selection cache impact **AddFamily** performance. The selection cache mitigates the tagging overhead in **AddFamily** because it caches the generated key schedule for $\tau_{k_{r,j}^{\text{sel}}}$, reducing the overhead of tag generation. Without the selection cache, computing tags increases **AddFamily** latency by up to $2\times$.

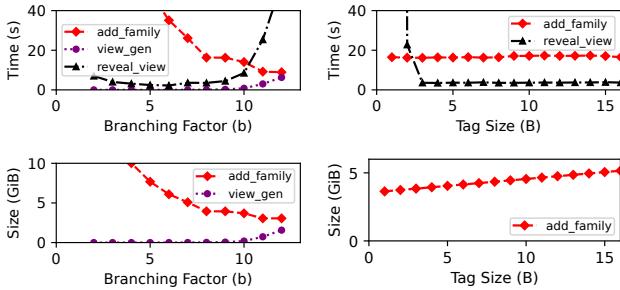
The selection cache can bring performance gains even at small sizes (Fig. 15). The reason is that, even if not all unique values of the predicate can fit in the selection cache simultaneously, datasets may be distributed in a way such that only a few unique values constitute most of the occurrences. Our LRU replacement policy results in the most commonly occurring values usually being represented in the cache.



(a) Output of EncryptTable.

(b) Output of AddFamily.

Figure 16: Membrane’s size overhead.

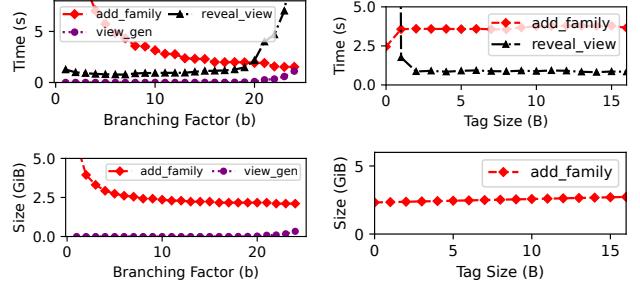
Figure 17: Varying branching factor and tag size, **rwe_ineq_and**. Branching factors are in bits; $b = 8$ is a branching factor of $2^8 = 256$.

The best selection cache size varies depending on the data distribution. For example, at our default selection cache size of 512, the selection cache actually reduces AddFamily performance for the **dropoff_pickup** workload. At a larger selection cache size of 8192, however, performance gains are realized (not shown in Fig. 15).

7.3 Compression and Size Overheads

We always encrypt data in *uncompressed* form; because encrypted data cannot be compressed, Membrane’s encryption leads to size overhead. The output of EncryptTable is similar in size to the *uncompressed* plaintext data, but an order of magnitude bigger than the *compressed* Parquet input (Fig. 16a). The size overhead of AddFamily relative to EncryptTable (due to proj./sel./tagging columns) is typically $\approx 1.5 \times$ or less, but up to $2 \times$ (Fig. 16b). Even including AddFamily’s overheads, loss of compression dominates size overheads.

Our implementation uses compressed Parquet to encode plaintext data (input to EncryptTable and output of RevealView). For encrypted data (which do not benefit from compression), we use Arrow’s serialization format (ipc), which does not have compression but is faster to (de)serialize than Parquet. To show the benefit of using ipc, Fig. 19 compares three serialization formats (Parquet with compression, Parquet without compression, and ipc), with the input and output stored on the local SSD in the same format. Note that the time to read/write the file (including serialization) is significant compared to the cryptographic processing time, and that ipc tends to be faster than Parquet.

Figure 18: Varying branching factor and tag size, **rwe_ineq_obs**. Branching factors are in bits; $b = 8$ is a branching factor of $2^8 = 256$.

7.4 Space/Time Trade-Offs

Two factors present a space/time trade-off: (1) truncating tags and (2) choosing the branching factor. Fig. 17 and Fig. 18 measure these trade-offs for **rwe_ineq_obs** and **rwe_ineq_and**. We choose these workloads because they use inequalities and have canonical forms with many predicates, so they are most sensitive to the branching factor and tag length.

Decreasing the tag length makes the output of AddFamily smaller (because tags are smaller), but decreasing it too much makes RevealView slower due to frequent false-positive tag matches. Increasing the branching factor makes the AddFamily faster and its output smaller because it reduces the number of predicates in canonical form, but increasing it too much makes RevealView much slower since it must check more keys for each row. Importantly, intermediate values (e.g., branching factor of $2^8 = 256$, 4-byte tags) appropriately balance this trade-off and result in all-around good performance.

7.5 Scalability to Multiple CPU Cores

We scale Membrane’s backend to multiple cores using large, multi-partition datasets. We use a `Standard_E104ids_v5` instance and omit the `store_sales` table, which does not fit in memory. As in §7.1, we measure the time for Membrane’s backend to process in-memory serialized data, and exclude the time to read the input or write the output. Because Membrane uses a separate *process* per core, partitions are *statically* assigned to threads/cores, for this benchmark only.

See Fig. 20. Membrane’s protocols scale linearly across multiple physical cores and sockets. While our system had 52 cores, we also measure performance with 104 threads, to use logical CPU cores (hyperthreading); as expected their benefit is less than physical cores. Results for RevealView are noisier than for AddFamily, possibly due to static partitioning, variable runtime for partitions, and NUMA effects.

7.6 End-to-End Performance

Planning. With a branching factor of 256, planning typically takes less than 10 ms—the longest time is for **rwe_ineq_and**, at ≈ 300 ms—and consumes less than 100 MiB of memory.

Interacting with cloud storage. We now measure RevealView when using Membrane’s orchestrator, both with (“*ppln*”) and without (“*stg*”) the orchestrator’s pipelining. We use `Standard_E104ids_v5` and materialize decrypted views

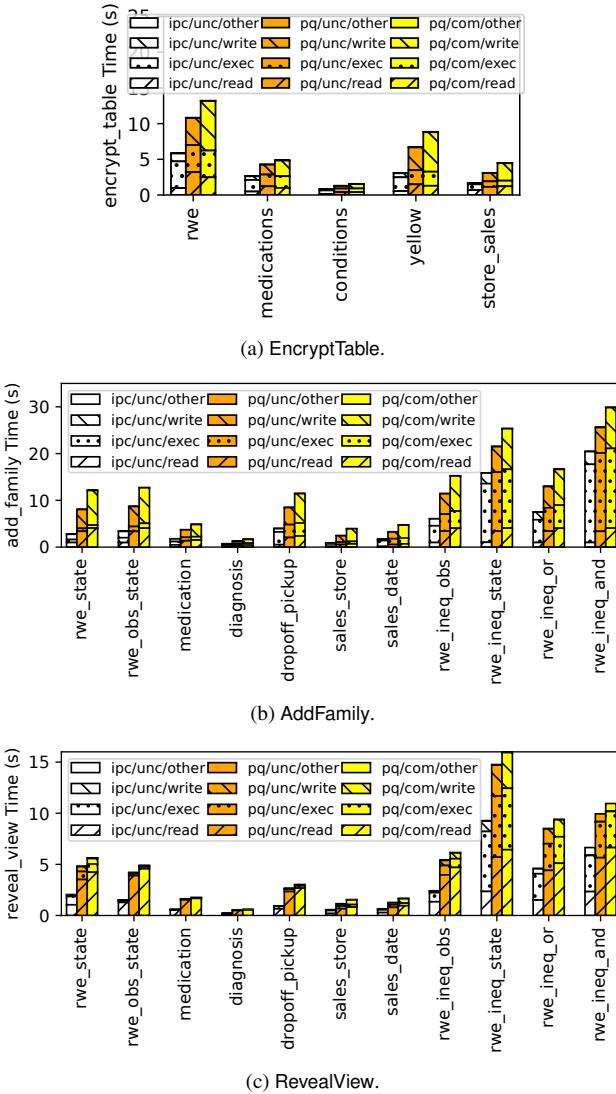


Figure 19: Membrane total runtime in single-core setting, including reading/writing files from local temp SSD, for different data formats.

on the local SSD. Fig. 22 shows results when processing *all* partitions of a table to hide access patterns (§3). While pipelining is generally faster, disabling it shows a clearer breakdown into individual components. Membrane’s backend is a significant fraction of overall runtime for some workloads (e.g., **rwe_ineq_state**), but fetching partitions from storage usually dominates. Fig. 23 shows results for EncryptTable and AddFamily—compared to Fig. 22, this graph has an additional component, *upload time*, since EncryptTable and AddFamily upload their output to storage. Fig. 21 shows results when using *fil* to only process partitions that contain view contents. We use **rwe_ineq_obs** and **sales_date**, as their data are in a contiguous range of partitions due to how the tables are sorted. Using *fil* is much faster (compare y-axes of Fig. 22 & Fig. 21) and reduces the relative overhead of fetching partitions, at the cost of revealing access patterns. Both figures show time normalized by the full *compressed* table size.

Protocol	Dec. Latency	Dec. Thrpt (rows/s/core)
IPE	≈ 4 ms	≈ 250
IBE	≈ 1 ms	≈ 1,000
Membrane	≈ 0.002 ms	≈ 500,000

Table 2: Estimates for single-predicate view (e.g., **rwe_state**).

Interactive analytics. We ran Membrane in a Databricks-hosted Spark cluster in an interactive notebook. The cluster has 16 machines, each with 8 CPUs and 64 GiB RAM, similar to LHBench’s evaluation setup [59]. Our baseline is to run PySpark-SQL in the standard way, providing it with the dataset’s cloud storage URL. To run Membrane, we call *map* on Spark RDDs to run the orchestrator to have workers process disjoint sets of partitions, and use internal Spark APIs to convert output partition files at workers into a Spark dataframe.

For each dataset (Synthea, NYC Taxi, and LHBench), we obtained analytical SQL queries (Appendix C). We separately measure the time to the result of the first SQL query, and the time to run the remaining SQL queries. After decrypting an AC view with Membrane, we call *persist* so that Spark caches it in memory for subsequent queries. We did not call *persist* in the baseline, so Spark’s query optimizer can co-optimize the AC view and query.

The results are in Fig. 24. On the first query, the baseline fetches plaintext data from cloud storage. Membrane fetches the result of *AddFamily*, which is larger due to loss of compression, and then runs *RevealView* on the data. Thus, Membrane increases the time until the first query result by up to 20×. Subsequent analytical queries execute on the cached result of *RevealView* and perform similarly to the baseline. For **dropoff_pickup**, they were actually *faster* with Membrane. This may be because Membrane locally materializes the view.

7.7 Comparison to Other Systems

There is no existing cryptographic system with the same functionality as Membrane. The closest is CryptDB’s multi-principal design [88, §4], but (1) the design does not support *private, encrypted* access control attributes, and (2) the public CryptDB code does not even support multiple principals. Thus, we are forced to consider baselines that do not exactly match Membrane’s functionality and/or security.

Existing Cryptographic Schemes. First, we consider Inner Product Encryption (IPE) [65], capable of conjunctions and disjunctions, to encrypt each row. IPE removes the restriction that the `SELECT` clause must include fields used in the `WHERE` clause (§4.1), as IPE decryption hides which `OR` clauses match. Second, we consider Identity-Based Encryption (IBE) [19]; an ID-hiding IBE scheme can be used instead of Membrane’s selection keys for each predicate. These designs still rely on Membrane’s view families, canonical form, and planner.

We consider schemes [67, 72] based on prime-order bilinear groups. We estimate their decryption time by counting the number of bilinear group operations and multiplying by the

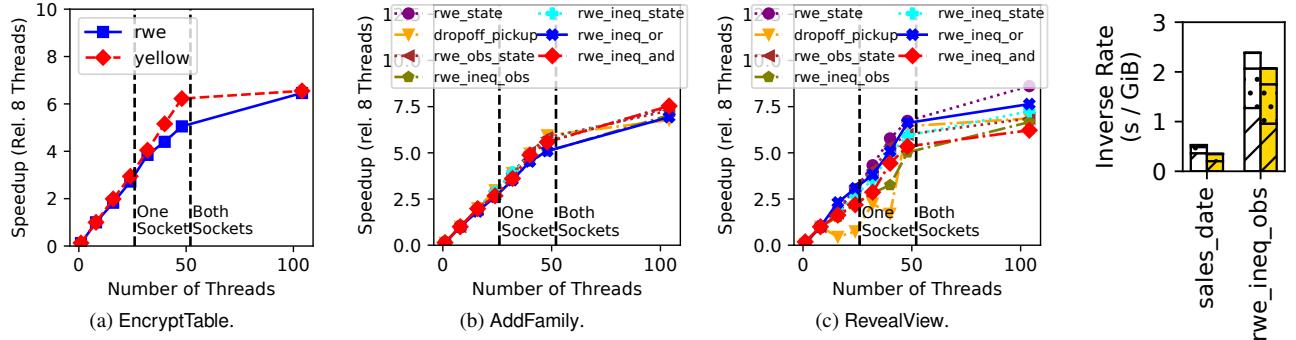


Figure 20: Multi-core scalability. Speedup is relative to 8 threads (e.g., linear scaling at 48 cores would be 6×). Figure 21: Effect of fil.

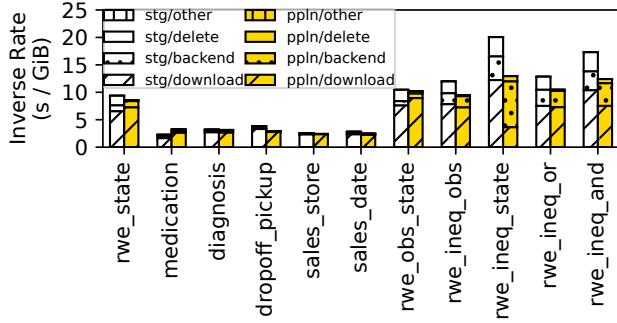


Figure 22: End-to-end breakdown for RevealView.

measured cost of those operations for an efficient bilinear group implementation [69, Table 1]. For Membrane, we use results from Fig. 12, dividing the total decryption time for the most populous state by the number of matching rows for that state. See Table 2. Our design based on symmetric-key primitives allows Membrane’s backend to decrypt rows up to three orders of magnitude faster than using off-the-shelf IPE/IBE. This is separate from Membrane’s key-hiding tags, which allow skipping rows that cannot be decrypted.

Fig. 22 shows that transferring data over the network can dominate the overall time. Table 2 clarifies that this is because Membrane uses only fast symmetric-key cryptography. In an IPE- or IBE-based design, decryption time at the client would dominate runtime, particularly for views with high selectivity.

Trusted AC Server. We consider the Trusted AC Server strawman (§2.1). Membrane’s benefits relative to this strawman are in security—the AC server is an online, central point of attack. We expect the strawman to outperform Membrane.

Building an efficient AC Server required forgoing API compatibility with Azure Data Lake Storage. For example, azcopy scans files ahead of time to determine their lengths before downloading them in chunks; supporting this at the AC service would require downloading the file and computing its view just to know the length of the resulting file, and doing so again to provide the file contents. Thus, existing tools like azcopy cannot directly interface with our AC server, requiring us to implement a new client to fetch files.

We ran the client and AC server on E104ids_v5 instances; results are in Fig. 25 (see Fig. 22 for comparison). First,

the AC service performs better than Membrane, as expected (green bars). But, for uncompressed datasets, its performance is comparable to Membrane with pipelining; this suggests that the AC server’s performance benefits are mostly explained by the fact that it can work with compressed data. Second, the CPU time at the AC servers was significant, highlighting the need to provision large amounts of compute to deploy an AC server. Third, 104 concurrent requests (one per logical core) were generally sufficient to achieve the best performance. Finally, for Membrane and the AC server, performance is better for more selective views. For Membrane, this is due to key-hiding tags; for the AC server, this may be because more selective views are smaller to transfer from AC server to client.

Our AC server uses TLS, but data are not encrypted in cloud storage. As noted in §2, one could encrypt cloud storage and have the AC server use a symmetric key to decrypt files on demand. We expect the performance impact to be small.

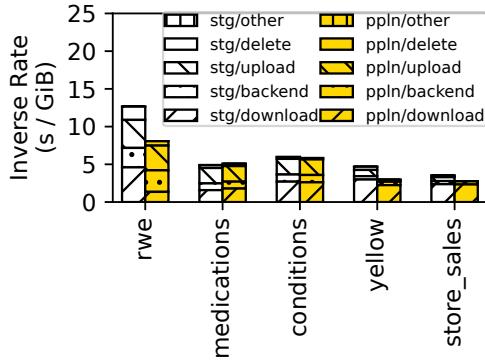
8 Related Work

As discussed in §1, EFSes [7, 17, 46, 62, 66, 87, 98, 100, 101] enforce access policies based on *public* file paths, at *file*-granularity. In contrast, Membrane enforces access policies based on *private*, encrypted, data values, at *cell*-granularity.

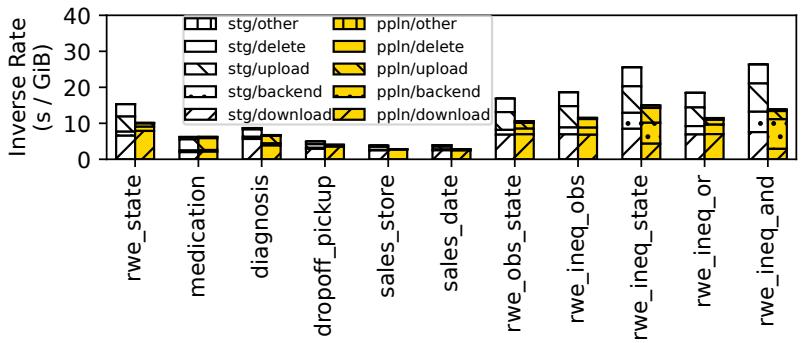
As discussed in §1, EDBs [13, 28, 30, 32, 41, 53, 63, 64, 83, 84, 86, 88, 94, 95] and ESSes [30, 33, 35, 40, 60, 61, 84, 97, 99] solve a different problem than Membrane. Whereas EDBs (respectively, ESSes) compute the *encrypted* result of a SQL query (respectively, keyword search query) without a decryption key, Membrane produces the *plaintext* result of an AC view using a decryption key for that view. Specifically, EDBs and ESSes have two important shortcomings compared to Membrane. First, they restrict clients to only issuing certain kinds of queries—those that can be executed cryptographically at the server. Second, they usually do not support multi-client access control. We discuss the few exceptions below.

The few EDBs that support access control generally do so based on public, unencrypted attributes [28, 54, 69, 93, 105]. For Query 1 (§1), these techniques would require exposing each row’s State. In contrast, Membrane encrypts all cells and enforces access control based on encrypted cell contents.

CryptDB considers multi-principal access control [88, §4].



(a) EncryptTable.



(b) AddFamily.

Figure 23: End-to-end breakdown of total runtime.

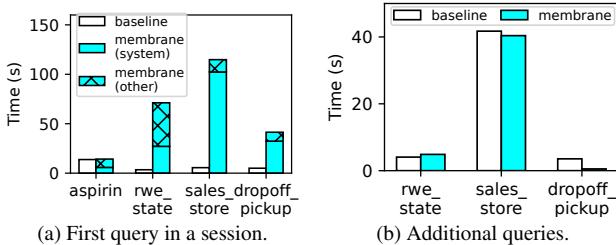


Figure 24: Membrane’s overhead in an interactive session.

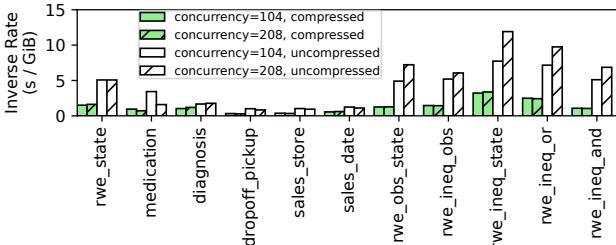


Figure 25: AC server performance (see also Fig. 22).

Compared to Membrane, it has limitations: (1) it is designed for access policies based on *public data*, requiring principals with access to a row to be listed in that row in *plaintext*, (2) it does not support inequalities or *ANDs*, and (3) it does not protect users logged in during a compromise.

MONOMI [102] is designed to support any SQL query. It works by offloading operations that it cannot support cryptographically to the client. The client, like Membrane’s compute, is trusted to decrypt the data and see them in plaintext. Unlike Membrane, MONOMI does not support access control.

Blind Seer [84] supports access control, but only for a single client—a single access policy is applied to *all* queries—and splits the server into multiple trust domains (S and IS).

Curtmola et al. [35] propose an ESS, but it is limited to single-word queries and lacks access control. Protocols based on OXT [31, 61] or structured encryption [33, 63, 64] support more complex queries, but still not access control. MC-OXT [60] extends OXT to multiple clients, but in a weaker threat model than Membrane—an adversary who has compromised *both* the storage server *and* some clients can bypass

access control. OXT-based schemes also use more complex and slower cryptography than Membrane, whose symmetric-key, high-throughput design is key to data lakes.

A line of work provides cryptographic access control for XML [2, 15, 16, 77]. Unlike Membrane, these works do not support relational data or data-dependent inequalities. They may also leak information about documents’ tree structure.

Parquet modular encryption [4, 45] provides coarse-grained, *column-level* encryption and access control. In contrast, Membrane provides data-dependent, *row/cell-level* encryption.

DJoin [79, §5.2] rewrites queries to an intermediate language (IL). DJoin provides DP, not access control, so its IL has a different structure than Membrane-canonical form.

Some EDBs [8, 12, 104, 108] use a Trusted Execution Environment (TEE) like Intel SGX. However, TEEs are complex hardware artifacts that are difficult to secure and are vulnerable to side-channel attacks [26, 27, 85, 103]. Membrane moves storage outside of the TCB by placing trust in cryptography.

9 Discussion and Conclusion

Data lakes departed from DBMSes by separating compute and storage. This enables independent scaling of compute and storage, and flexible use of data analysis frameworks (e.g., Spark, Pandas) instead of SQL.

This paper shows that the data lake architecture, originally motivated by scalability and flexibility, actually has positive implications for security. Specifically, data lakes organize compute and storage into *separate trust domains*. This allows Membrane to focus on removing storage entirely from the TCB via fine-grained, data-dependent, cryptographically-enforced access control, without also having to remove compute entirely from the TCB. Thus, Membrane can process analytical queries in plaintext, thereby retaining the flexibility of unencrypted data lakes and supporting off-the-shelf frameworks.

This does not, by itself, render EDBs and ESSes insufficient for Membrane’s purpose. The actual reason why existing EDBs and ESSes are not viable alternatives to Membrane is **access control**; they either do not provide access control

at all, or provide weaker access control than Membrane (§8). If an EDB or ESS were to provide access control, it could be adapted to data lakes by applying Membrane’s system model—specifically, by having data scientists query the EDB/ESS for their AC views and then analyze the results in plaintext (§1).

That said, Membrane’s techniques can be applied to existing single-client EDBs/ESSes to enable access control. This is possible because Membrane’s goal, namely *cryptographic access control*, is orthogonal to EDBs’ and ESSes’ goal, namely *query processing over encrypted data*.

As a concrete example, consider CryptDB [88]. CryptDB includes a multi-user access control design [88, §4], in which a user’s data is decrypted at the proxy/server when a user logs in, and the proxy/server is trusted to securely delete the users’ key and decrypted data when they log out. As discussed in §8, CryptDB’s design cannot support access policies based on private, encrypted data. To remedy this, we can apply Membrane’s techniques, by using Membrane’s cryptographic protocol as the outermost layer of onion encryption. Specifically, each cell is encrypted with CryptDB’s scheme as usual, and then Membrane is used to encrypt the resulting ciphertexts, while ensuring that $g(\text{row})$ is evaluated on *plaintext row data*. When a user logs in, she sends her Membrane view keys to the proxy/server, which decrypts the matching cells to obtain the CryptDB ciphertexts. CryptDB can then process queries over these ciphertexts as it is designed to do. When the user logs out, the CryptDB proxy/server is trusted to delete the users’ view keys and the decrypted CryptDB ciphertexts, analogous to CryptDB’s current access control.

Similarly to our construction for CryptDB in the previous paragraph, Membrane’s backend can encrypt data in an ESS for access control. But Membrane has even deeper connections to ESSes. The $g(\text{row}) \text{ IN } ?x$ clauses in Membrane-canonical form (§4) represent a generalized form of *keyword search*, the functionality that ESSes provide. Rows in Membrane correspond directly to documents in an ESS, and the expression $g(\text{row})$ corresponds to a generalization of keywords by which documents can be queried. Thus, Membrane’s planner, which rewrites complex queries into keyword searches, can actually enable ESSes to process more complex queries.

Given that Membrane’s backend and planner also apply to EDBs and ESSes, why did we focus on data lakes in this paper? First, it shows that Membrane’s techniques are useful independently of the encrypted query processing in EDBs and ESSes and results in a simpler system design for Membrane. Second, it shows that Membrane’s techniques are broadly applicable, as they allow data scientists to use off-the-shelf analytics frameworks and do not restrict the analytical query set as EDBs/ESSes do. Third, it shows that Membrane’s design is *synergistic* with current trends in data analytics systems, specifically the separation between compute and storage that has come to define the data lake paradigm.

Finally, Membrane’s amortized performance overhead can

be small, as it only requires decrypting data at the *start* of an interactive session. Thus, we are hopeful that Membrane’s techniques can help protect sensitive data used for analytics.

Acknowledgments

We would like to thank Joey Gonzalez and Matei Zaharia for formative conversations in the early stages of this research project. This work is supported by NSF CISE Expeditions #CCF-1730628, NSF GRFP #DGE-1752814, and gifts from Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computation soundness of formal encryption). In *TCS*. Springer-Verlag Berlin Heidelberg, 2000.
- [2] Martín Abadi and Bogdan Warinschi. Security analysis of cryptographically controlled access to XML documents. *Journal of the ACM (JACM)*, 55(2):1–29, 2008.
- [3] Apache Parquet. Apache Parquet. <https://parquet.apache.org/>. Accessed: March 17, 2023.
- [4] Apache Parquet. Parquet modular encryption. <https://parquet.apache.org/docs/file-format/data-pages/encryption/>. Accessed: April 2, 2024.
- [5] Apache Software Foundation. Apache Arrow | Apache Arrow. <https://arrow.apache.org/>. Accessed: April 10, 2023.
- [6] Apache Software Foundation. Apache ORC • high-performance columnar storage for Hadoop. <https://orc.apache.org/>. Accessed: April 10, 2023.
- [7] Apple Inc. iCloud data security overview - Apple Support. <https://support.apple.com/en-us/HT202303>. Accessed: April 16, 2023.
- [8] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. Transaction processing on confidential data using Cipherbase. In *ICDE*. IEEE, 2015.
- [9] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*. CIDR, 2021.
- [10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng,

- Romer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*. ACM, 2015.
- [11] Microsoft Azure. NYC taxi & limousine commission - yellow taxi trip records. <https://learn.microsoft.com/en-us/azure/open-datasets/dataset-taxi-yellow?tabs=azureml-opendatasets>, 2022.
- [12] Sumeet Bajaj and Radu Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *SIGMOD*. ACM, 2011.
- [13] Johnes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: Secure querying for federated databases. *VLDB*, 10(6), 2017.
- [14] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT*. Springer-Verlag Berlin Heidelberg, 2001.
- [15] Elisa Bertino and Elena Ferrari. Secure and selective dissemination of xml documents. *TISSEC*, 5(3), 2002.
- [16] Elisa Bertino, Gabriel Ghinita, Ashish Kamra, et al. Access control for databases: Concepts and systems. *Foundations and Trends® in Databases*, 3(1–2):1–148, 2011.
- [17] Matt Blaze. A cryptographic file system for Unix. In *CCS*. ACM, 1993.
- [18] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In *EUROCRYPT*. Springer, 2005.
- [19] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*. Springer, Berlin, Heidelberg, 2001.
- [20] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *TCC*. Springer, Berlin, Heidelberg, 2011.
- [21] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*, chapter 5. cryptobook.us, 2020.
- [22] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*, chapter 4. cryptobook.us, 2023.
- [23] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*, chapter 6. cryptobook.us, 2023.
- [24] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*, chapter 5. cryptobook.us, 2023.
- [25] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*, chapter 2. cryptobook.us, 2023.
- [26] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*. USENIX, 2017.
- [27] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*. USENIX, 2018.
- [28] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. TimeCrypt: Encrypted data stream processing at scale with cryptographic access control. In *NSDI*. USENIX, 2020.
- [29] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*. ACM, 2015.
- [30] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Ros, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*. Internet Society, 2014.
- [31] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roșu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*. Springer, Berlin, Heidelberg, 2013.
- [32] David Cash, Ruth Ng, and Adam Rivkin. Improved structured encryption for SQL databases via hybrid indexing. In *Applied Cryptography and Network Security*. Springer International Publishing, 2021.
- [33] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*. Springer Berlin Heidelberg, 2010.
- [34] Ericka Chickowski. Leaky buckets: 10 worst Amazon S3 breaches. *Bitdefender*, 2018. <https://businessinsights.bitdefender.com/worst-amazon-breaches>. Accessed: March 17, 2023.
- [35] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS*. ACM, 2006.
- [36] Databricks. Create a dynamic view. <https://docs.databricks.com/data-governance/unity-catalog>

- [og/create-views.html#dynamic-view](https://www.databricks.com/doc/api/latest/og/create-views.html#dynamic-view). Accessed: April 17, 2023.
- [37] Databricks. Delta Lake on Databricks. <https://www.databricks.com/product/delta-lake-on-databricks>. Accessed: March 18, 2023.
- [38] Databricks. Immuta - Databricks. <https://www.databricks.com/partners/immuta>. Accessed: March 18, 2023.
- [39] Databricks. Privacera - Databricks. <https://www.databricks.com/partners/privacera>. Accessed: March 18, 2023.
- [40] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *OSDI*. USENIX, 2020.
- [41] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *S&P*. IEEE, 2022.
- [42] Cynthia Dwork. Differential privacy. In *ICALP*. Springer, Berlin, Heidelberg, 2006.
- [43] Zachary Friedman. Considerations for data access in the Lakehouse, 2021. https://youtu.be/aUD4Z_U_nWmM. Accessed: March 18, 2023.
- [44] Abhinav Garg and Tianyi Huang. Databricks Lakehouse platform governance and security fundamentals. https://www.databricks.com/session_na21/databricks-lakehouse-platform-governance-and-security-fundamentals, Accessed: March 19, 2023.
- [45] Gidon Gershinsky. Efficient analytics on encrypted data. In *SYSTOR*. ACM, 2018.
- [46] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*. Internet Society, 2003.
- [47] Google. BigLake: Unify data lakes & data warehouses | Google Cloud. https://cloud.google.com/big_lake. Accessed: March 18, 2023.
- [48] Google. Protocol buffers documentation. <https://protobuf.dev/>. Accessed: April 10, 2023.
- [49] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-Based Encryption for fine-grained access control of encrypted data. In *CCS*. ACM, 2006.
- [50] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*. ACM, 2016.
- [51] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*. ACM, 2017.
- [52] Paul Allen Grubbs. *Breaking and Building Encrypted Databases*. PhD thesis, Cornell University, 2019.
- [53] Hakan Hacıgümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*. ACM, 2002.
- [54] Isabelle Hang, Florian Kerschbaum, and Ernesto Damiani. ENKI: Access control for encrypted query processing. In *SIGMOD*. ACM, 2015.
- [55] Synthetic Health. Synthea. <https://github.com/synthetichealth/synthea>, 2023.
- [56] Brenner Heintz and Denny Lee. Productionizing machine learning with Delta Lake - Databricks blog. <https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html>. Accessed: March 22, 2023.
- [57] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghosstor: Toward a secure data-sharing system from decentralized trust. In *NSDI*. USENIX, 2020.
- [58] Sean Jacobs and Matt Vogt. Eisai's secret to data access control in Databricks lakehouse with SQL analytics. <https://youtu.be/FOfRGg41RVI>. Accessed: March 18, 2023.
- [59] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. Analyzing and comparing lakehouse storage systems. In *CIDR*. CIDR, 2023.
- [60] Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *CCS*. ACM, 2013.
- [61] Charanjit Jutla and Sikhar Patranabis. Efficient searchable symmetric encryption for join queries. In *ASIACRYPT*. Springer-Verlag, 2022.
- [62] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*. USENIX, 2003.
- [63] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In *ASIACRYPT*. Springer-Verlag, 2018.
- [64] Seny Kamara, Tarik Moataz, Stan Zdonik, and Zheguang Zhao. An optimal relational database encryption scheme. Cryptology ePrint Archive, Paper

- 2020/274, 2020. <https://eprint.iacr.org/2020/274>.
- [65] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2008.
- [66] Keybase. Keybase. <https://keybase.io/>. Accessed: April 16, 2023.
- [67] Intae Kim, Seong Oun Hwang, Jong Hwan Park, and Chanil Park. An efficient predicate encryption with constant pairing computations and minimum costs. *Transactions on Computers*, 65(10), 2016.
- [68] Jeremy Kirk. Verizon breach: 6 million customer accounts exposed. *Bank Info Security*, 2017. <https://www.bankinfosecurity.com/verizon-breach-6-million-customer-accounts-exposed-a-10107>. Accessed: March 19, 2023.
- [69] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. In *USENIX Security*. USENIX, 2019.
- [70] Matthias Lècuyer, Riley Spahn, Kiran Vodrahalli, Roxana Geambasu, and Daniel Hsu. Privacy accounting and quality control in the Sage differentially private ML platform. In *SOSP*. ACM, 2019.
- [71] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*. USENIX, 2004.
- [72] Benoît Libert and Jean-Jacques Quisquater. Identity based encryption without redundancy. In *ACNS*. Springer, Berlin, Heidelberg, 2005.
- [73] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *OSDI*. USENIX, 2010.
- [74] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *VLDB*, 3(1), 2010.
- [75] Microsoft. Data Lake | Microsoft Azure. <https://azure.microsoft.com/en-us/solutions/data-lake/>. Accessed: March 18, 2023.
- [76] Microsoft. Parquet format in Azure Data Factory and Azure Synapse Analytics. <https://learn.microsoft.com/en-us/azure/data-factory/format-parquet>. Accessed: December 10, 2024.
- [77] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Proceedings 2003 VLDB Conference*, pages 898–909. Elsevier, 2003.
- [78] Junta Nakai and Anna Cuisia. Guide to financial services sessions at Data + AI Summit 2022. <https://www.databricks.com/blog/2022/05/31/guide-to-financial-services-sessions-at-data-a-summit-2022.html>. Accessed: March 19, 2023.
- [79] Arjun Narayan and Andreas Haeberlen. DJoin: Differentially private join queries over distributed databases. In *OSDI*. USENIX, 2012.
- [80] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS*. ACM, 2015.
- [81] Michael Ortega and Michael Sanku. Guide to health-care & life sciences sessions at Data + AI Summit 2022. <https://www.databricks.com/blog/2022/06/14/guide-to-healthcare-life-sciences-sessions-at-data-ai-summit-2022.html>. Accessed: March 19, 2023.
- [82] Pandas. pandas - Python data analysis library. <https://pandas.pydata.org/>. Accessed: April 10, 2023.
- [83] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*. USENIX, 2016.
- [84] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steven Bellovin. Blind Seer: A scalable private dbms. In *S&P*. IEEE, 2014.
- [85] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *S&P*. IEEE, 2011.
- [86] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *VLDB*, 12(11), 2019.
- [87] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX ATC*. USENIX, 2011.
- [88] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*. ACM, 2011.

- [89] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, and Amrish Kumar. The Cosmos big data platform at Microsoft: Over a decade of progress and a decade to look forward. *VLDB*, 14(12), 2021.
- [90] Mark Raasveldt and Hannes Mühleisen. DuckDB: an embeddable analytical database. In *SIGMOD*. ACM, 2019.
- [91] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter 4. McGraw-Hill, 3 edition, 2003.
- [92] Rodman Ramezanian. It’s plane to see—unsecured servers can put lives at stake: How an exposed S3 bucket exposed 3TB worth of sensitive airport data, 2022. <https://www.skyhighsecurity.com/en-us/about/resources/intelligence-digest/unsecured-servers-can-put-lives-at-stake.html>. Accessed: March 17, 2023.
- [93] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. DBMask: Fine-grained access control on encrypted relational databases. In *CODASPY*. ACM, 2015.
- [94] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure sharing of partially homomorphic encrypted IoT data. In *Sensys*. ACM, 2017.
- [95] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. Talos: Encrypted query processing for the Internet of Things. In *Sensys*. ACM, 2015.
- [96] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *SIGCOMM*. ACM, 2015.
- [97] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *S&P*. IEEE, 2000.
- [98] SpiderOak. Space cybersecurity solutions for hybrid space | SpiderOak. <https://spideroak.com/>. Accessed: April 16, 2023.
- [99] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*. Internet Society, 2014.
- [100] Sync.com, Inc. Sync | secure cloud storage, file sharing and document collaboration. <https://www.sync.com/>. Accessed: April 16, 2023.
- [101] Tresorit. End-to-end encrypted cloud storage for businesses | tresorit. <https://tresorit.com/>. Accessed: April 16, 2023.
- [102] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *VLDB*, 6(5), 2013.
- [103] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAXe: How SGX fails in practice. <https://sgaxeattack.com/>, 2020.
- [104] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. StealthDB: a scalable encrypted database with full SQL support. *Privacy Enhancing Technologies*, 2019(3), 2019.
- [105] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*. USENIX, 2016.
- [106] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.
- [107] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*. USENIX, 2016.
- [108] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*. USENIX, 2017.
- [109] Zscaler. Anatomy of a cloud breach: How 100 million credit card numbers were exposed. *Zscaler*, 2021. <https://www.zscaler.com/resources/white-papers/capital-one-data-breach.pdf>. Accessed: March 17, 2023.
- [110] Corey Zwart, Itai Weiss, and Steve Touw. Turning fan data into an asset. <https://youtu.be/DYtEmdr3k0c>. Accessed: April 5, 2023.

A Full Backend Protocol Description

A.1 EncryptTable

Input:

- Partition (index p) of table t
 - $m_{r,c}$ denotes value of cell at row r and column c
- Table key k^{tab} (chosen randomly; same for all partitions)

Outputs: Partition (index p) of encrypted table t'

Algorithm: Sample $k^{\text{tab}} \leftarrow \{0,1\}^\lambda$. For p in $1 \dots n^{\text{part}}$:

- For each row index r in $1 \dots n_p^{\text{row}}$:
 - Derive row key $k_r := \text{PRF}(k^{\text{tab}}, p || r)$
 - For each column c in $1 \dots n^{\text{col}}$:
 - Derive cell key $k_{r,c} := \text{PRF}(k_r, c)$
 - The value of t' at r, c (in partition p) is $\text{OTE}(k_{r,c}, m_{r,c})$

A.2 AddFamily

Inputs:

- Encrypted partition (index p) of table t
- Table key k^{tab}
- Family key k^{fam} (chosen randomly; same for all partitions)
- Indices $c_1, \dots, c_{n^{\text{proj}}}$ of columns projected from t
- Predicate functions $g_1 \dots g_{n^{\text{pred}}}$ used for selection

Outputs: Projection/selection/tagging columns for partition

Algorithm: For each row index r in $1 \dots n_p^{\text{row}}$:

- Compute the value of the projection column.
 - Define $k_r := \text{PRF}(k^{\text{tab}}, p || r)$, as in EncryptTable.
 - Compute k_r^{proj} :
 - If $n^{\text{proj}} = 1$, then $k_r^{\text{proj}} = \text{PRF}(k_r, c_1)$
 - If $n^{\text{proj}} = n^{\text{col}}$, then $k_r^{\text{proj}} = k_r$
 - Else, sample $k_r^{\text{proj}} \leftarrow \{0,1\}^\lambda$
 - The value of the projection column at row r is:
 - If $n^{\text{proj}} = 1$ or $n^{\text{proj}} = n^{\text{col}}$: $\text{PRF}(k_r^{\text{proj}}, 0)$
 - Else: $\text{Enc}(k_r^{\text{proj}}, k_{r,c_1} || \dots || k_{r,c_{n^{\text{proj}}}}) || \text{Enc}(k_r^{\text{proj}}, 0)$
- Compute the value of the selection column.
 - For each predicate j in $1 \dots n^{\text{pred}}$:
 - Derive $k_j^{\text{pred}} := \text{PRF}(k^{\text{fam}}, j)$
 - Derive $k_{r,j}^{\text{sel}} := \text{PRF}(k_j^{\text{pred}}, g_j(m_{r,1} || \dots || m_{r,n^{\text{col}}}))$
 - The value of the selection column at row r is $\text{Enc}(\text{PRF}(k_{r,1}^{\text{sel}}, 0), k_r^{\text{proj}}) || \dots || \text{Enc}(\text{PRF}(k_{r,n^{\text{pred}}}^{\text{sel}}, 0), k_r^{\text{proj}})$
- Compute the value of the tagging column.
 - For each predicate j in $1 \dots n^{\text{pred}}$:
 - Derive $\tau_{k_{r,j}^{\text{sel}}} := \text{PRF}(k_{r,j}^{\text{sel}}, p)$.
 - Derive tag $_{k_{r,j}^{\text{sel}}} := \text{PRF}(\tau_{k_{r,j}^{\text{sel}}}, \text{count}_{k_{r,j}^{\text{sel}}})$, and truncate it to the desired length.
 - Tagging column at row r contains $\text{tag}_{k_{r,1}^{\text{sel}}} || \dots || \text{tag}_{k_{r,n^{\text{pred}}}^{\text{sel}}}$

A.3 ViewGen

Inputs:

- View family key k^{fam}
- List of wildcard values X^j for each predicate g_j in the view

Outputs: View key set K^{view}

Algorithm:

- Define $K^{\text{view}} := []$

- For j in $1 \dots n^{\text{pred}}$:

- Define $K^{\text{pred}} := []$ and define $k_j^{\text{pred}} := \text{PRF}(k^{\text{fam}}, j)$
- For x_i^j in X^j :
 - Let $k_{j,i}^{\text{view}} := \text{PRF}(k_j^{\text{pred}}, x_i^j)$ and append $k_{j,i}^{\text{view}}$ to K^{pred}
- Append K^{pred} to K^{view}

A.4 RevealView

Inputs:

- t^{fam} , the output of AddFamily on table t . The partition index is denoted p .
- View key set K^{view}

Outputs: Decrypted view over t

Algorithm: For each partition p in t^{fam} :

- For each key $k_{j,i}^{\text{view}}$, define $\text{count}_{k_{j,i}^{\text{view}}}$ and initialize it to 0.
- For each $k_{j,i}^{\text{view}}$ in K^{view} : Define $\tau_{k_{j,i}^{\text{view}}} := \text{PRF}(k_{j,i}^{\text{view}}, p)$.
- For each key $k_{j,i}^{\text{view}}$, compute its NET as $\text{PRF}(\tau_{k_{j,i}^{\text{view}}}, 0)$, truncated to the desired length.
- Define a map N' from each NET to a tuple of the corresponding key $k_{j,i}^{\text{view}}$ and the predicate index j .
- For each row r in t^{fam} :
 - For each tag in the tagging column:
 - Look up the decryption key $k_{j,i}^{\text{view}}$ in N' . If not found, continue (i.e., skip to the next tag in the inner loop).
 - Do $\text{DecryptRow}(t^{\text{fam}}[r], k_{j,i}^{\text{view}}, j)$. If unsuccessful, continue (i.e., skip to the next tag in the inner loop).
 - Increment $\text{count}_{k_{j,i}^{\text{view}}}$.
 - Compute new NET for $k_{j,i}^{\text{view}}$: $\text{PRF}(\tau_{k_{j,i}^{\text{view}}}, \text{count}_{k_{j,i}^{\text{view}}})$, truncated to the desired length.
 - Update N' to reflect the new NET (i.e., remove mapping for old NET and add mapping for new NET).

DecryptRow

Inputs:

- Encrypted row (index r) of t^{fam}
- Candidate key k^{sel} and matched predicate index j

Outputs: Decrypted row of the view $t', t'[r]$

Algorithm:

- Decrypt the j th entry of the selection column to reveal k^{proj} : $k^{\text{proj}} := \text{Dec}(\text{PRF}(k^{\text{sel}}, 0), \text{SelCol}[j])$.
- Decrypt projection column with k^{proj} to reveal cell keys:
 - If the projection column has one element, check that $\text{PRF}(k^{\text{proj}}, 0) = \text{ProjCol}[1]$. Otherwise, check that $\text{Dec}(k^{\text{proj}}, \text{ProjCol}[2]) = 0$. If the check fails, abort.
 - If the view includes only one column c_1 (so $n^{\text{proj}} = 1$), then decrypt the value of $t'[r]$ as $\text{Dec}(k^{\text{proj}}, t^{\text{fam}}[r][c_1])$.
 - Else, if the view includes all columns (so $n^{\text{proj}} = n^{\text{col}}$), then re-derive the cell keys from k^{proj} . For each non-family column c included in the view:
 - Derive the cell key $k_{r,c} := \text{PRF}(k^{\text{proj}}, c)$.
 - Decrypt the value of $t'[r][c]$ as $\text{Dec}(k_{r,c}, t^{\text{fam}}[r][c])$.
 - Else, decrypt the cell keys $k_{r,c_1} || \dots || k_{r,c_{n^{\text{proj}}}} = \text{Dec}(k^{\text{proj}}, \text{ProjCol}[2])$. For each column c in the view:
 - Decrypt the value of $t'[r][c]$ as $\text{Dec}(k_{r,c}, t^{\text{fam}}[r][c])$.

B Cryptographic Treatment of Membrane

We now provide a cryptographic treatment of Membrane and its security guarantees (described informally in §2.2).

B.1 System Model

We define an EDL scheme as follows.

Definition 1. An EDL (“encrypted data lake”) scheme is a tuple of four algorithms:

- $\text{EncryptTable}(t, k^{\text{tab}}) \rightarrow t'$
- $\text{AddFamily}(t, k^{\text{tab}}, \text{fam}, k^{\text{fam}}) \rightarrow t'$
- $\text{ViewGen}(\text{view}, k^{\text{fam}}) \rightarrow k^{\text{view}}$
- $\text{RevealView}(t, k^{\text{view}}) \rightarrow t'$

The syntax matches Fig. 2, except that k^{tab} and k^{fam} are inputs, not outputs (as in §5). With the above syntax, the caller is assumed to sample k^{tab} uniformly at random before calling `EncryptTable`, and to sample k^{fam} uniformly at random before calling `AddFamily`. This is analogous to the caller sampling a symmetric key uniformly at random before invoking symmetric-key encryption.

Using this syntax allows our cryptographic formalism to model the case where `ViewGen` is called for a view family before `EncryptTable` or `AddFamily` are called (i.e., where a view key is generated for a view family before that view family is instantiated in a table). In that sense, this syntax is more general than the one given in Fig. 2. Fig. 2 presents the API as it does because it is more suggestive of Membrane’s intended use case, and is more similar to the API actually provided by our implementation.

We require the intuitive notion of “completeness” that `RevealView` indeed produces the same result as materializing the view in plaintext. We define completeness as follows.

Definition 2. An EDL scheme is said to be complete if for any table t , AC view family f , and AC view v where $v \in f$, the following holds:

If we run $\text{EncryptTable}(t, k^t) \rightarrow t'$ and $\text{AddFamily}(t', k^t, f, k^f) \rightarrow t''$ (and possibly other `AddFamily` operations on the table) and $\text{ViewGen}(v, k^f) \rightarrow k^v$ then $\text{RevealView}(t'', k^v) = v(t)$

where $v(t)$ denotes the view v applied to the table t , and where k^t and k^f are sampled uniformly at random.

B.2 Security Definition

In defining security, we consider AC view families with the constraint that an AC view family must `SELECT` all columns referenced in its `WHERE` clause. This means that the set of cell positions described by an AC view includes both the cell positions that the view reveals and the cell positions that the `WHERE` clause references for rows where cells are revealed (as discussed in §4.1).

Our main security definition is for a notion that we call *selective security*.

Definition 3 (Selective Security EDL Game). Selective security for an EDL scheme is defined in terms of the following game between an adversary \mathcal{A} and challenger \mathcal{C} .

Initialization. \mathcal{A} chooses the schemas of u relations and the size n_i for each relation (for $1 \leq i \leq u$), where n_i includes the number of partitions and size of each partition. \mathcal{A} also chooses a set P of cell positions. Each cell position is a tuple (i, p, r, c) where i is the index of a relation, p is a partition ID, r is the row index, and c is the column index. \mathcal{A} “declares” u , its chosen schemas, n_i ($\forall 1 \leq i \leq u$), and P , by sending them to \mathcal{C} .

Phase 1. \mathcal{A} repeatedly issues queries to \mathcal{C} . There are two types of queries that \mathcal{A} may make:

- \mathcal{A} asks \mathcal{C} to instantiate a view family in one of the relations. \mathcal{C} samples the corresponding k^{fam} uniformly at random but does not give k^{fam} to \mathcal{A} .
- \mathcal{A} asks \mathcal{C} to generate a key for a view v belonging to a view family specified in a previous query (identified by the ID of the earlier query). \mathcal{C} generates the view key by calling `ViewGen` on the k^{fam} generated in the earlier query and gives the resulting view key to \mathcal{A} .

Challenge. \mathcal{A} chooses two u -length tuples of relations, \mathcal{R}_0 and \mathcal{R}_1 . The choice is subject to the following restrictions:

1. A cell in \mathcal{R}_0 and a cell in \mathcal{R}_1 , both at position (i, p, r, c) , must contain identical data if $(i, p, r, c) \notin P$, and must contain data of the same length if $(i, p, r, c) \in P$.
2. Each requested view v must describe the same set of cell positions, denoted $v^{\mathcal{R}}$, whether applied to \mathcal{R}_0 or \mathcal{R}_1 , and those cell positions must be disjoint from P (i.e., it must hold that $v^{\mathcal{R}} \cap P = \emptyset$).

\mathcal{A} sends \mathcal{R}_0 and \mathcal{R}_1 to \mathcal{C} . \mathcal{C} chooses a random bit $b \xleftarrow{\$} \{0, 1\}$. Then, it encrypts \mathcal{R}_b according to Membrane’s protocol, by calling `EncryptTable` on each table (with a randomly sampled k^{tab} for each table). For each view family specified in the queries in Phase 1, it calls `AddFamily` to instantiate the view family in the encrypted \mathcal{R}_b , using the k^{tab} for the specified table and the k^{fam} for that view family. Finally, it sends the resulting encrypted relations, t_1, \dots, t_u , to \mathcal{A} .

Phase 2. \mathcal{A} can issue additional queries of the same form as those in Phase 1, subject to Constraint #2 in the Challenge phase. When \mathcal{A} requests a view family, \mathcal{C} instantiates the view family by calling `AddFamily` and responds to \mathcal{A} with the updated table t' right away. As in Phase 1, \mathcal{A} does not reveal k^{fam} to \mathcal{C} .

Guess. \mathcal{A} outputs $b' \in \{0, 1\}$, and wins the game if $b = b'$. The advantage of the adversary \mathcal{A} is defined as $|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}|$.

Now, we define selective security in terms of that game.

Definition 4. An EDL scheme is selectively secure if, for any non-uniform probabilistic polynomial-time adversary \mathcal{A} , it holds that \mathcal{A} ’s advantage in the Selective Security EDL Game (Definition 3) is negligible.

B.2.1 Discussion of Our Security Definition

We refer to our notion of security as *selective security* because we require \mathcal{A} to *select*, ahead of time, which cells to attack. The cells that the adversary chooses to attack are those in P , and the adversary is not allowed to query cells that they are choosing to attack (i.e., any set P_v of queried cells positions must be disjoint from P). This is analogous to *selective security* definitions used in the context of Identity-Based Encryption [18] and Attribute-Based Encryption [49], where the adversary must select, ahead of time, which ID or set of attributes to attack, and may not query the private key corresponding to that ID or those attributes.

In a fully *adaptive* notion of security, the adversary would not have to declare the set P of cell positions up front, and would not be restricted by P in the Queries phase. Because our selective security definition requires the adversary to declare P during the Initialization phase, with the analogous restrictions in the Queries phase, it is weaker (i.e., protects against a weaker adversary) than an adaptive security definition would be. We use a selective notion of security rather than a fully adaptive one because adaptive notions of security are difficult to achieve in practice.

B.2.2 Comparison to Static Security

A commonly used weaker alternative to adaptive security is static security. In static security definitions, the adversary \mathcal{A} commits up front to a sequence of queries that she will make to \mathcal{C} . While our selective security definition may be weaker than fully adaptive security, our selective security definition is at least as strong as a static security definition. The intuition for this is that, given a sequence of queries declared up front by a static adversary, one can compute the set P of cell positions to declare up front in the Selective Security EDL Game.

To present this argument more formally, we first provide a formal definition of static security. This allows us to prove via a reduction that selective security is at least as strong as static security.

Definition 5 (Static Security EDL Game). *Static security for an EDL scheme is defined in terms of the following game.*

Initialization. \mathcal{A} chooses the schemas of u relations and the size n_i for each relation (for $1 \leq i \leq u$), where n_i includes the number of partitions and size of each partition. \mathcal{A} chooses F and V , defined as follows:

- F is a list of view families, including tables (specified by index) in which \mathcal{C} must instantiate each of them.
- V maps each view family in F to a list of views in that view family, whose keys \mathcal{C} must generate and reveal to \mathcal{A} .

Each cell position is a tuple (i, p, r, c) where i is the index of a relation, p is a partition ID, r is the row index, and c is the column index. \mathcal{A} “declares” u , its chosen schemas, n_i ($\forall 1 \leq i \leq u$), F , and V , by sending them to \mathcal{C} .

Challenge. \mathcal{A} chooses two u -length tuples of relations, \mathcal{R}_0 and \mathcal{R}_1 , subject to the following restrictions:

1. A cell in \mathcal{R}_0 and a cell in \mathcal{R}_1 , both at position (i, p, r, c) , must contain identical data if any view $v \in V$ describes (i, p, r, c) , and must contain data of the same length if no view in V describes (i, p, r, c) .

2. Each view $v \in V$ must describe the same set of cell positions, denoted $v^{\mathcal{R}}$, whether applied to \mathcal{R}_0 or \mathcal{R}_1 .

\mathcal{A} sends \mathcal{R}_0 and \mathcal{R}_1 to \mathcal{C} . \mathcal{C} chooses a random bit $b \xleftarrow{\$} \{0, 1\}$. Then, it encrypts \mathcal{R}_b according to Membrane’s protocol, by calling *EncryptTable* on each table (with a randomly sampled k^{tab} for each table). For each view family in F , it invokes *AddFamily* to instantiate the view family in the encrypted \mathcal{R}_b , using the k^{tab} for the specified table and a randomly sampled k^{fam} for each view family. Finally, it sends the resulting encrypted relations, t_1, \dots, t_u , to \mathcal{A} . \mathcal{C} does not send the family keys k^{fam} for the view families in F to \mathcal{A} . \mathcal{C} invokes *ViewGen* for each view in V (using the corresponding family key k^{fam}) and sends the resulting view keys to \mathcal{A} .

Guess. \mathcal{A} outputs $b' \in \{0, 1\}$, and wins the game if $b = b'$. The advantage of the adversary \mathcal{A} is defined as $|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}|$.

Now, we define static security in terms of that game.

Definition 6. An EDL scheme is statically secure if, for any non-uniform probabilistic polynomial-time adversary \mathcal{A} , it holds that \mathcal{A} ’s advantage in the Static Security EDL Game (Definition 5) is negligible.

Using these definitions, we state and prove, in formal terms, that selective security is at least as strong as static security.

Theorem 1. If an EDL scheme is selectively secure (Definition 4), then it is statically secure (Definition 6).

Proof. We prove this statement using contraposition—we show that, if a non-uniform probabilistic polynomial-time adversary $\mathcal{A}_{\text{static}}$ can win the Static Security EDL Game (Definition 5) with non-negligible probability for the EDL scheme, then there exists a non-uniform probabilistic polynomial-time adversary \mathcal{A}_{sel} that can win the Selective Security EDL Game (Definition 3) with non-negligible probability for the EDL scheme. We do so using a reduction, constructing \mathcal{A}_{sel} as an algorithm that uses $\mathcal{A}_{\text{static}}$ as a black box. The following paragraphs describe \mathcal{A}_{sel} .

For the **Initialization** phase, \mathcal{A}_{sel} observes the output of $\mathcal{A}_{\text{static}}$ in its Initialization and Challenge phases and “declares” the same values for u , the relations’ schemas, and n_i ($\forall 1 \leq i \leq u$), by sending them to \mathcal{C} . \mathcal{A}_{sel} also “declares” the set P , computed as follows. Because \mathcal{R}_0 and \mathcal{R}_1 have the same number of relations, and corresponding relations have the same size, partitioning, and schema, the set of all possible cell positions is the same for both; let T refer to this set of all possible cell positions. For each view $v \in V$, \mathcal{A}_{sel} computes $v^{\mathcal{R}}$, the set of cell positions revealed by that view (which both games require to be the same in both \mathcal{R}_0 and \mathcal{R}_1). Then \mathcal{A}_{sel} computes Q , a set of cell positions, as

$$Q = \bigcup_{v \in V} v^{\mathcal{R}}$$

and computes P as $P = T \setminus Q$. In effect, P is the set of cell positions *not* covered by any of the views chosen by $\mathcal{A}_{\text{static}}$.

In **Phase 1**, \mathcal{A}_{sel} issues queries corresponding to the values F and V output by $\mathcal{A}_{\text{static}}$ during its Initialization phase.

In the **Challenge** phase, \mathcal{A}_{sel} sends \mathcal{C} the same values for \mathcal{R}_0 and \mathcal{R}_1 as $\mathcal{A}_{\text{static}}$ would send to its challenger. If $\mathcal{A}_{\text{static}}$ outputs valid \mathcal{R}_0 and \mathcal{R}_1 that satisfy the requirements of the Challenge phase in the Static Security EDL Game, then the same \mathcal{R}_0 and \mathcal{R}_1 , output by \mathcal{A}_{sel} , will satisfy the requirements of the Challenge phase in the Selective Security EDL Game:

1. P is chosen such that the cell positions in P are exactly those that are not described by any view $v \in V$. Therefore, “any view $v \in V$ describes (i, p, r, c) ” is equivalent to $(i, p, r, c) \notin P$. Thus, if \mathcal{R}_0 and \mathcal{R}_1 satisfy Constraint #1 in the Static Security EDL Game, then they satisfy Constraint #1 in the Selective Security EDL Game.
2. Constraint #2 of the Selective Security EDL Game has two conditions. The first condition is identical to Constraint #2 in the Static Security EDL Game. The second condition holds because of how we constructed P —by definition, $Q \not\subseteq P$, and $\forall v \in V v^{\mathcal{R}} \subseteq Q$, so $\forall v \in V v^{\mathcal{R}} \cap P = \emptyset$. Therefore, if \mathcal{R}_0 and \mathcal{R}_1 satisfy Constraint #2 in the Static Security EDL Game, then they satisfy Constraint #2 in the Selective Security EDL Game.

\mathcal{A}_{sel} receives the encrypted relations from \mathcal{C} and sends them to $\mathcal{A}_{\text{static}}$. \mathcal{A}_{sel} also gives $\mathcal{A}_{\text{static}}$ the view keys it obtained from \mathcal{C} in Phase 1.

In **Phase 2**, \mathcal{A}_{sel} does not issue any queries.

As its **Guess**, \mathcal{A}_{sel} outputs the same bit b' as $\mathcal{A}_{\text{static}}$. The information that \mathcal{A}_{sel} gave $\mathcal{A}_{\text{static}}$ is distributed identically to what $\mathcal{A}_{\text{static}}$ would receive from a challenger in the Static Security EDL Game who chose the same value of b as \mathcal{C} did. Therefore, \mathcal{A}_{sel} has the same advantage in the Selective Security EDL Game as $\mathcal{A}_{\text{static}}$ does in the Static Security EDL Game. \square

B.3 Preliminaries

Membrane’s protocol depends on three cryptographic primitives: a pseudorandom function (PRF), an encryption scheme Enc , and a one-time encryption scheme OTE . Here, we discuss these cryptographic primitives, their security guarantees, and how we instantiate them in our Membrane implementation.

B.3.1 Pseudorandom Functions

A *pseudorandom function* (PRF) is a deterministic function that accepts as input a key k and a message x ; we denote its application as $\text{PRF}(k, x)$. A formal treatment of PRFs is given by Boneh and Shoup [22, Definition 4.2].

In our Membrane implementation, we instantiate PRFs in two different ways.

The first way is to simply use the AES block cipher as a PRF. This is a valid approach because the Switching Lemma [22, Theorem 4.4] guarantees that if AES is a secure block cipher (pseudorandom permutation), then it is also a

secure PRF. This technique is very efficient; in particular, when using AES-128 as a PRF, the key size is the same as the block size, allowing the PRF’s output to be used directly as the key to another PRF invocation. Unfortunately, this approach limits the size of the PRF input to the block size (e.g., 16 bytes in the case of AES-128), so we cannot instantiate every PRF in Membrane’s protocol in this way.

The second way, which supports arbitrary-size inputs, is to use CBC-MAC, instantiated with the AES block cipher, together with prepending the input length to the input. This approach is valid because the CBC-MAC construction, alone, produces a prefix-free PRF [23, Theorem 6.3], so when it is used with a prefix-free encoding of the input, as is obtained when prepending the input’s length, it produces a fully secure PRF [23, Theorem 6.8].

B.3.2 Symmetric-Key Encryption

Membrane requires a symmetric-key encryption scheme Enc with two properties: CPA-security and key-privacy. A formal treatment of CPA-security is given by Boneh and Shoup [24, Theorem 5.2]. Bellare et al. [14] coin the term *key-privacy* in the public-key setting; here we require an analogous property in the symmetric-key setting. Abadi and Rogaway [1, Definition 2] provide a formal security definition for a symmetric-key encryption scheme that is both CPA-secure and key-private; they refer to key-private encryption as “which-key concealing” and to such CPA-secure and key-private symmetric-key encryption schemes as “type-1 secure.”

In our Membrane implementation, we instantiate the symmetric-key encryption scheme Enc by using the AES block cipher in CTR mode. Abadi and Rogaway [1, Section 4.4] explain that CTR-mode encryption is indeed “type-1 secure.”

B.3.3 One-Time Symmetric-Key Encryption

Membrane makes use of a one-time symmetric-key encryption scheme OTE . It does so for efficiency; it would be correct and secure to instantiate OTE in exactly the same way as we instantiated Enc , but the idea is that OTE can be instantiated in a more efficient way. This is possible because, unlike Enc , OTE need not support key reuse. Specifically, OTE is semantically secure, as defined by Boneh and Shoup [25, Definition 2.2], but not necessarily CPA-secure.

In our Membrane implementation, we instantiate OTE by using AES in CTR mode for messages longer than the key (just as in Enc), but using the one-time pad scheme [25, Example 2.2], which is more efficient, for short messages.

B.4 Security Guarantee and Proof of Security

Now, we state and sketch a proof of a theorem describing Membrane’s security.

Theorem 2 (Membrane’s Security Guarantee). *If Membrane is instantiated with a secure PRF, a CPA-secure and key-private encryption scheme Enc , and a semantically secure*

one-time encryption scheme OTE, then Membrane is a selectively secure EDL scheme under Definition 4.

Proof Sketch. We make a hybrid argument, presenting a sequence of hybrid games that present the same interface to \mathcal{A} as the Selective Security EDL Game but in which \mathcal{C} replies with differently formed messages. The first hybrid \mathcal{H}_0 is identical to the Selective Security EDL Game, and the final hybrid \mathcal{H}_* is one where \mathcal{A} 's advantage is 0 by construction. We argue that for any two adjacent hybrid games in the sequence \mathcal{H}_j and \mathcal{H}_{j+1} , the difference in \mathcal{A} 's advantage is negligible. Because the number of hybrid games is polynomial in the security parameter λ , this implies that \mathcal{A} 's advantage in the game \mathcal{H}_0 is negligible, as desired.

We now present the sequence of hybrid games. In each step except the last, we change only how \mathcal{C} responds to queries. In some cases, these changes are localized to only how certain rows are processed in an AddFamily operation for certain AC view families. We refer to such combinations of rows and AC view families as *critical combinations*. Specifically, an AC view family f and a row (p, r) form a critical combination if f `SELECTs` some column (i, c) such that $(i, p, r, c) \in P$. Our hybrid games in this proof sketch should be interpreted as key stages; between each pair of stages are multiple hybrid games, where only one instance of a cryptographic primitive is changed at a time.

Hybrid \mathcal{H}_0 . This game is exactly the Selective Security EDL Game (i.e., Definition 3).

Hybrid \mathcal{H}_1 . This is the same as \mathcal{H}_0 except that we replace every row key with a truly random value. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of a PRF keyed by the truly random table key, together with the fact that each row key for a table is generated using a different input to the PRF (namely $r \parallel c$).

Hybrid \mathcal{H}_2 . This is the same as \mathcal{H}_1 except that we replace each predicate key k_j^{pred} with a truly random value. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of a PRF keyed by the truly random view family key k^{fam} , together with the fact that each predicate key k_j^{pred} for a given AC view family is generated using a different input to the PRF (namely j).

Hybrid \mathcal{H}_3 . This is the same as \mathcal{H}_2 except that we replace PRFs keyed on each predicate key k_j^{pred} with truly random functions. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of a PRF keyed by the truly random predicate keys. Note that, although the PRFs keyed on k_j^{pred} are now replaced with uniformly random functions and \mathcal{C} uses these random function to derive the selection keys $k_{r,j}^{\text{sel}}$, this is *not* equivalent to \mathcal{C} sampling each row's selection key uniformly at random. Specifically, selection keys may still repeat across rows; this is because multiple rows may have the same value of $g_j(\text{row})$, which is used as the input to the PRF.

Hybrid \mathcal{H}_4 . This is the same as \mathcal{H}_3 except that, for critical

combinations of AC view families and rows, we replace PRFs keyed on selection keys $k_{r,j}^{\text{sel}}$ with truly random functions when \mathcal{C} runs AddFamily. Specifically, we associate each selection key with its own truly random function, and replace each PRF invocation keyed on that selection key with an invocation of its truly random function. This impacts the derivation of encryption keys for the selection layer (used to encrypt the projection keys) and the derivation of each partition's tagging key, for those critical combinations. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of PRFs keyed by the selection keys, which are sampled randomly (due to \mathcal{H}_3). This requires that, for these critical combinations, the selection keys are not revealed to \mathcal{A} ; this holds because \mathcal{A} may only request view keys for which $P_v \cap P = \emptyset$, which implies that none of \mathcal{A} 's requested view keys includes a $k_{j,x}^{\text{view}}$ matching a selection key for a critical combination as part of \mathcal{A} 's requested view keys. In order for this to hold, it is important that views must `SELECT` columns referenced in their `WHERE` clause (i.e., P_v includes the cell positions that the `WHERE` clause references for rows where cells are revealed), as mentioned in §4.1.

Hybrid \mathcal{H}_5 . This is the same as \mathcal{H}_4 except that, for critical combinations of AC view families and rows, we replace tags with random strings when running AddFamily. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of a PRF keyed on the tagging keys, which are the result of a random function due to \mathcal{H}_4 ; while the tagging keys for a predicate may repeat among rows, the combination of tagging key and counter is always different when generating each tag (i.e., the counters guarantee that the PRF invocation to generate the tag is always performed with a different counter when the key is reused).

Hybrid \mathcal{H}_6 . This is the same as \mathcal{H}_5 except that, for critical combinations of AC view families and rows, we replace each encryption of the projection key in the selection column with an encryption of a “zero string” under the same key, when \mathcal{C} runs AddFamily. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the CPA-security of the encryption scheme Enc.

Hybrid \mathcal{H}_7 . This is the same as \mathcal{H}_6 except that, for critical combinations of AC view families and rows, we replace encryptions of “zero strings” in the selection column with encryptions of “zero strings” *under random keys*. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the key-privacy of the encryption scheme Enc. Observe that, at this hybrid stage, the ciphertexts in the selection column, for critical combinations, are entirely independent of the data chosen by \mathcal{A} at cell positions in P .

Hybrid \mathcal{H}_8 . This is the same as \mathcal{H}_7 except that, for critical combinations of AC view families and rows, we replace the encryption of cell keys with an encryption of “zero strings” of the same length. In some optimized cases, the encryption of cell keys is not present; for such view families, this hy-

brid step changes nothing compared to \mathcal{H}_6 . The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the CPA security of the encryption scheme when using k_r^{proj} as the encryption key (since, for critical combinations, the encryption of k_r^{proj} has been replaced with an encryption of a “zero string,” meaning that k_r^{proj} is never revealed to \mathcal{A}).

Hybrid \mathcal{H}_9 . This is the same as \mathcal{H}_8 except that, for cell positions in P , we replace every cell key with a truly random value. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the security of a PRF keyed by the truly random row keys (after the \mathcal{H}_1 step). Importantly, after the \mathcal{H}_7 step, row keys are no longer revealed to \mathcal{A} for critical combinations, even in the optimized case where $k_r^{\text{proj}} = k_r$. Note that, for this to work in the optimized case where the projection key is the row key, replacing the encryption of 0 with a PRF invocation at 0 is crucial.

Hybrid \mathcal{H}_{10} . This is the same as \mathcal{H}_9 except that cells at positions in P , which normally contain OTE encryptions of cell data, are changed to instead contain OTE encryptions of “zero strings” of the same length. The difference in \mathcal{A} 's advantage between the previous stage and this stage is negligible because of the semantic security of the one-time encryption scheme OTE, together with the fact that the cell keys are random (due to \mathcal{H}_9) and never reused.

In \mathcal{H}_{10} , the data chosen by \mathcal{A} at cell positions in P have *no influence* on the values that \mathcal{C} gives to \mathcal{A} in response to any query; this is because the encryptions of those data have been replaced with encryptions of zero strings, and any keys that are derived from them have been replaced with random values. The distribution of data that \mathcal{C} gives to \mathcal{A} is identical whether $b = 0$ or $b = 1$; in particular, the data chosen by \mathcal{A} at cell positions outside of P are identical in \mathcal{R}_0 and \mathcal{R}_1 . Therefore, \mathcal{A} 's advantage in the \mathcal{H}_{10} game is 0. We take $\mathcal{H}_* = \mathcal{H}_{10}$, completing the proof sketch. \square

B.5 Discussion

The above covers the security of Membrane’s cryptographic backend protocol. Our implementation of Membrane uses a collision-resistant hash to support inequality operations on strings. It does so by hashing strings to integers, to leverage Membrane’s support for inequalities over integer quantities. This use of collision-resistant hashing is not covered by our formal definition above because collision-resistant hashing is merely used as a wrapper around Membrane’s cryptographic protocol; Membrane’s cryptographic backend protocol does not itself use collision-resistant hashing. In particular, any mechanism to map each string in a column to a unique integer would be sufficient for use with Membrane.

C Analytical Queries

```
--Based on Section 9.3 of
--https://ohdsi.github.io/TheBookOfOhdsi/SqlAndR.html.
--Query 1
SELECT AVG(DATEDIFF(DAY,
                      observation_period_start_date,
                      observation_period_end_date) / 365.25)
AS num_years
FROM (SELECT
       MIN(ENCOUNTER_START) AS observation_period_start_date,
       MAX(ENCOUNTER_STOP) AS observation_period_end_date
     FROM rwe_state
   GROUP BY OBSERVATION_PATIENT);

--Query 2
SELECT COUNT(DISTINCT OBSERVATION_PATIENT) FROM rwe_state;

--Query 3
SELECT MAX(YEAR(observation_period_end_date)
           - YEAR(date_of_birth)) AS max_age
FROM (SELECT
       MAX(ENCOUNTER_STOP) AS observation_period_end_date,
       first(PATIENT_BIRTHDATE) AS date_of_birth
     FROM rwe_state
   GROUP BY OBSERVATION_PATIENT);

--Query 4
WITH ages
AS (
  SELECT age,
         ROW_NUMBER() OVER (
           ORDER BY age
         ) order_nr
  FROM (
    SELECT YEAR(observation_period_end_date)
           - YEAR(date_of_birth) AS age
  FROM (SELECT
         MAX(ENCOUNTER_STOP)
           AS observation_period_end_date,
         first(PATIENT_BIRTHDATE) AS date_of_birth
       FROM rwe_state
     GROUP BY OBSERVATION_PATIENT)
    ) age_computed
  )
SELECT MIN(age) AS min_age,
       MIN(CASE
             WHEN order_nr < .25 * n
                 THEN 9999
             ELSE age
             END) AS q25_age,
       MIN(CASE
             WHEN order_nr < .50 * n
                 THEN 9999
             ELSE age
             END) AS median_age,
       MIN(CASE
             WHEN order_nr < .75 * n
                 THEN 9999
             ELSE age
             END) AS q75_age,
       MAX(age) AS max_age
  FROM ages
  CROSS JOIN (
    SELECT COUNT(*) AS n
    FROM ages
    ) population_size;
```

Listing 1: Analytical queries over the Synthea dataset.

```
--Query 5 (TPC-DS Query 3)
SELECT dt.d_year,
       item.i_brand_id      brand_id,
       item.i_brand         brand,
       sum(ss_ext_discount_amt) sum_agg
FROM   date_dim dt,
       sales_store store_sales,
       item
WHERE  dt.d_date_sk = store_sales.ss_sold_date_sk
       AND store_sales.ss_item_sk = item.i_item_sk
       AND item.i_manufact_id = 427
       AND dt.d_moy = 11
GROUP BY dt.d_year,
         item.i_brand,
         item.i_brand_id
ORDER BY dt.d_year,
         sum_agg DESC,
         brand_id
LIMIT 100;

--Query 6 (TPC-DS Query 7)
SELECT i_item_id,
       avg(ss_quantity) agg1,
       avg(ss_list_price) agg2,
       avg(ss_coupon_amt) agg3,
       avg(ss_sales_price) agg4
FROM   sales_store,
       customer_demographics,
       item,
       date_dim,
       promotion
WHERE  ss_sold_date_sk = d_date_sk
       AND ss_item_sk = i_item_sk
       AND ss_cdemo_sk = cd_demo_sk
       AND ss_promo_sk = p_promo_sk
       AND cd_gender = 'F'
       AND cd_marital_status = 'W'
       AND cd_education_status = '2_yr_Degree'
       AND ( p_channel_email = 'N'
              OR p_channel_event = 'N' )
       AND d_year = 1998
GROUP BY i_item_id ORDER BY i_item_id
LIMIT 100;

--Query 7 (TPC-DS Query 6)
SELECT a.ca_state state,
       count(*) cnt
FROM   customer_address a,
       customer c,
       sales_store s,
       date_dim d,
       item i
WHERE  a.ca_address_sk = c.c_current_addr_sk
       AND c.c_customer_sk = s.ss_customer_sk
       AND s.ss_sold_date_sk = d.d_date_sk
       AND s.ss_item_sk = i.i_item_sk
       AND d.d_month_seq = (SELECT DISTINCT (d_month_seq)
                            FROM   date_dim
                            WHERE  d_year = 1998
                                   AND d_moy = 7)
       AND i.i_current_price > 1.2 *
          (SELECT avg(j.i_current_price)
             FROM   item j
             WHERE  j.i_category = i.i_category)
GROUP BY a.ca_state
HAVING count(*) >= 10
ORDER BY cnt
LIMIT 100;
```

Listing 2: Analytical queries over the LHBench TPC-DS dataset.

```
--Query 8 (TPC-DS Query 4)
WITH year_total
AS (SELECT c_customer_id      customer_id,
           c_first_name     customer_first_name,
           c_last_name      customer_last_name,
           c_preferred_cust_flag
                           customer_preferred_cust_flag,
           c_birth_country   customer_birth_country,
           c_login          customer_login,
           c_email_address  customer_email_address,
           d_year           dyear,
           sum((ss_ext_list_price
                 - ss_ext_wholesale_cost
                 - ss_ext_discount_amt)
                  + ss_ext_sales_price) / 2) year_total,
's'                      sale_type
FROM   customer,
       sales_store store_sales,
       date_dim
WHERE  c_customer_sk = ss_customer_sk
       AND ss_sold_date_sk = d_date_sk
GROUP BY c_customer_id,
         c_first_name,
         c_last_name,
         c_preferred_cust_flag,
         c_birth_country,
         c_login,
         c_email_address,
         d_year
UNION ALL
SELECT c_customer_id      customer_id,
       c_first_name     customer_first_name,
       c_last_name      customer_last_name,
       c_preferred_cust_flag
                           customer_preferred_cust_flag,
       c_birth_country   customer_birth_country,
       c_login          customer_login,
       c_email_address  customer_email_address,
       d_year           dyear,
       sum((cs_ext_list_price
             - cs_ext_wholesale_cost
             - cs_ext_discount_amt)
                + cs_ext_sales_price) / 2) year_total,
'c'                      sale_type
FROM   customer,
       catalog_sales,
       date_dim
WHERE  c_customer_sk = cs_bill_customer_sk
       AND cs_sold_date_sk = d_date_sk
GROUP BY c_customer_id,
         c_first_name,
         c_last_name,
         c_preferred_cust_flag,
         c_birth_country,
         c_login,
         c_email_address,
         d_year
UNION ALL
SELECT c_customer_id      customer_id,
       c_first_name     customer_first_name,
       c_last_name      customer_last_name,
       c_preferred_cust_flag
                           customer_preferred_cust_flag,
       c_birth_country   customer_birth_country
```

```

        customer_birth_country,
c_login      customer_login,
c_email_address
            customer_email_address,
d_year          dyear,
Sum(( ( ws_ext_list_price
- ws_ext_wholesale_cost
- ws_ext_discount_amt)
+ ws_ext_sales_price ) / 2 )) year_total,
'w'           sale_type
FROM  customer,
web_sales,
date_dim
WHERE c_customer_sk = ws_bill_customer_sk
AND ws_sold_date_sk = d_date_sk
GROUP BY c_customer_id,
         c_first_name,
         c_last_name,
         c_preferred_cust_flag,
         c_birth_country,
         c_login,
         c_email_address,
         d_year)
SELECT t_s_secyear.customer_id,
       t_s_secyear.customer_first_name,
       t_s_secyear.customer_last_name,
       t_s_secyear.customer_preferred_cust_flag
FROM  year_total t_s_firstyear,
year_total t_s_secyear,
year_total t_c_firstyear,
year_total t_c_secyear,
year_total t_w_firstyear,
year_total t_w_secyear
WHERE t_s_secyear.customer_id = t_s_firstyear.customer_id
AND t_s_firstyear.customer_id
= t_c_secyear.customer_id
AND t_s_firstyear.customer_id
= t_c_firstyear.customer_id
AND t_s_firstyear.customer_id
= t_w_firstyear.customer_id
AND t_s_firstyear.customer_id
= t_w_secyear.customer_id
AND t_s_firstyear.sale_type = 's'
AND t_c_firstyear.sale_type = 'c'
AND t_w_firstyear.sale_type = 'w'
AND t_s_secyear.sale_type = 's'
AND t_c_secyear.sale_type = 'c'
AND t_w_secyear.sale_type = 'w'
AND t_s_firstyear.dyear = 2001
AND t_s_secyear.dyear = 2001 + 1
AND t_c_firstyear.dyear = 2001
AND t_c_secyear.dyear = 2001 + 1
AND t_w_firstyear.dyear = 2001
AND t_w_secyear.dyear = 2001 + 1
AND t_s_firstyear.year_total > 0
AND t_c_firstyear.year_total > 0
AND t_w_firstyear.year_total > 0
AND CASE
        WHEN t_c_firstyear.year_total > 0
        THEN t_c_secyear.year_total /
t_c_firstyear.year_total
        ELSE NULL
    END > CASE
        WHEN t_s_firstyear.year_total > 0 THEN
t_s_secyear.year_total /
t_s_firstyear.year_total
        ELSE NULL
    END
        AND CASE
        WHEN t_c_firstyear.year_total > 0
        THEN t_c_secyear.year_total /
t_c_firstyear.year_total
        ELSE NULL
    END > CASE
        WHEN t_w_firstyear.year_total > 0 THEN
t_w_secyear.year_total /
t_w_firstyear.year_total
        ELSE NULL
    END
    ORDER BY t_s_secyear.customer_id,
             t_s_secyear.customer_first_name,
             t_s_secyear.customer_last_name,
             t_s_secyear.customer_preferred_cust_flag
LIMIT 100;

```

Listing 3: Complex analytical query over the LHBench TPC-DS dataset.

```

--Based on https://learn.microsoft.com/en-us/sql/machine-learning/tutorials/demo-data-nyctaxi-in-sql.
--Query 9
SELECT DISTINCT passengerCount
, ROUND (SUM (fareAmount),0) as TotalFares
, ROUND (AVG (fareAmount),0) as AvgFares
FROM dropoff_pickup
GROUP BY passengerCount
ORDER BY AvgFares DESC;

--Query 10
SELECT * FROM dropoff_pickup LIMIT 10;

--Query 11
SELECT COUNT(*) FROM dropoff_pickup;

```

Listing 4: Analytical queries over the NYC Yellow Taxi dataset.