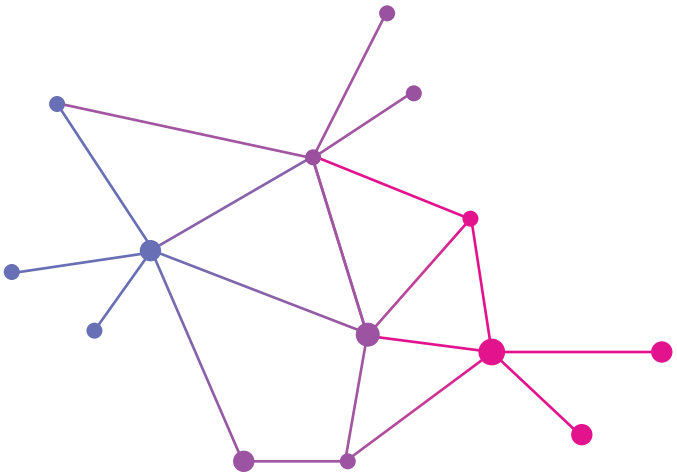
SEPTEMBER 2017

IMMUTA TECHNICAL OVERVIEW

The white paper provides a technical overview of the Immuta Data Management Platform including connecting your data, controlling data through complex policy enforcement, accessing data, connecting 3rd party tools and performance benchmarks.



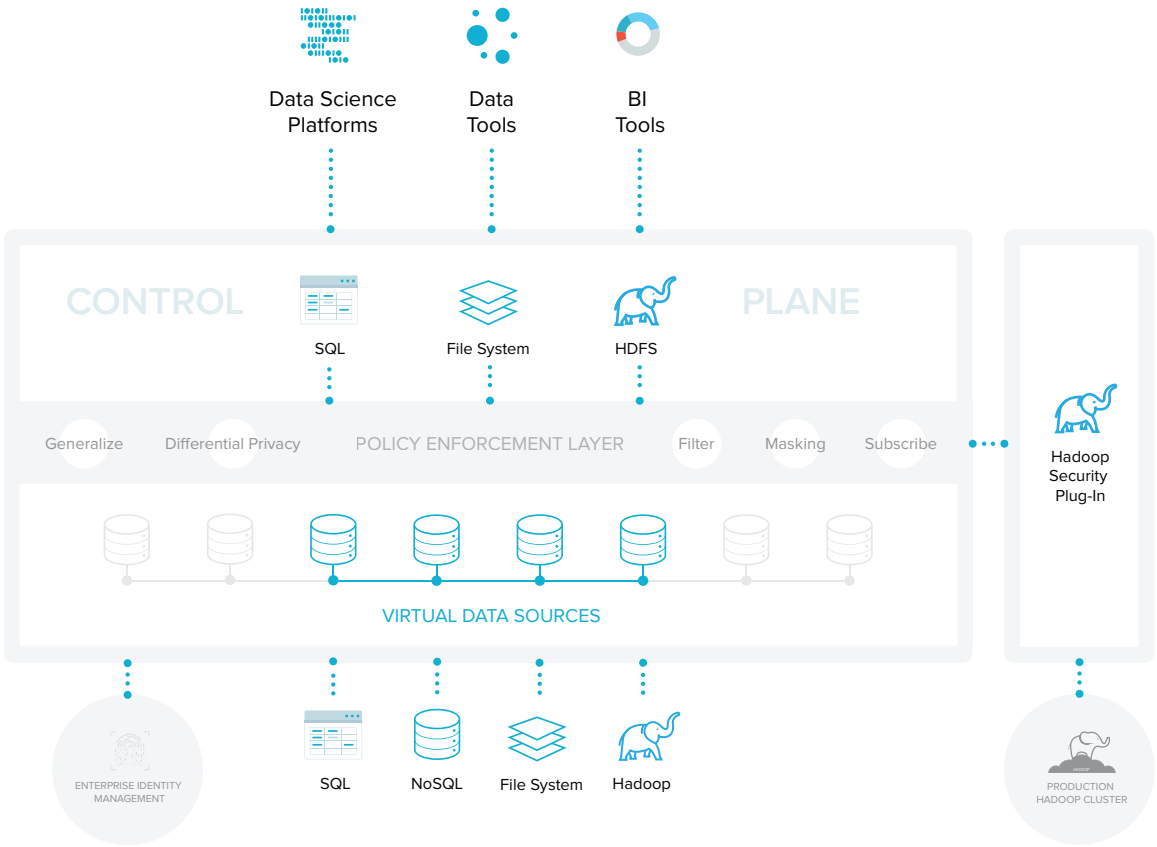
IMMUTA

IMMUTA

Table of Contents

About Immuta

Immuta's paradigm shift is the ability to unify your disparate data silos, virtually, allowing a consistent point for data access for all data analysis activities - the Immuta data control plane. The control plane dynamically protects the data with complex subscription and anonymization policies that are enforced on-the-fly based on the user accessing the data and the logic of the policy. This removes friction between data owners/compliance professionals and the data analysts/scientists by creating digital data exchanges compliant to your organization's regulations and providing complete visibility and flexibility into how policies are enforced and monitored.



Unification: All your data remains in-place, however, is virtually unified within Immuta for a single point of access.

Data Policies: Data is hidden, masked, redacted, anonymized on-the-fly in the control plane based on the attributes of the user accessing the data and the purpose they are acting under.

Authoritative Virtual Data Sources: Complex joins and data transformations can be exposed in Immuta as authoritative virtual data sources for downstream users.

Audit Consistency: All activities are audited in a consistent manner across your data silos from within the virtual layer. This simplifies audit log analysis and increases richness of your audit logs.

Data Access

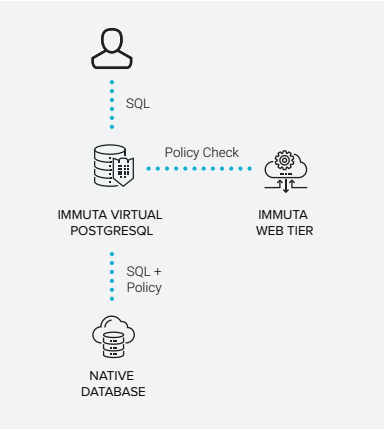
The Immuta data control plane does not require users to learn a new API or language to access data exposed there. Immuta plugs into existing tools and ongoing work, completely invisible to downstream consumers by exposing the data through foundational access patterns:

SQL: Users are provided a basic Immuta PostgreSQL connection. The tables within this connection represent all the connected data across your organization. Those tables, however, are virtual tables, completely empty until a query is run. At query time the SQL is proxied through the virtual Immuta table down to the native database while enforcing the policy on-the-fly. The Immuta SQL connection can be used within any Business Intelligence (BI) tool or integrated directly into code for interactive analysis.

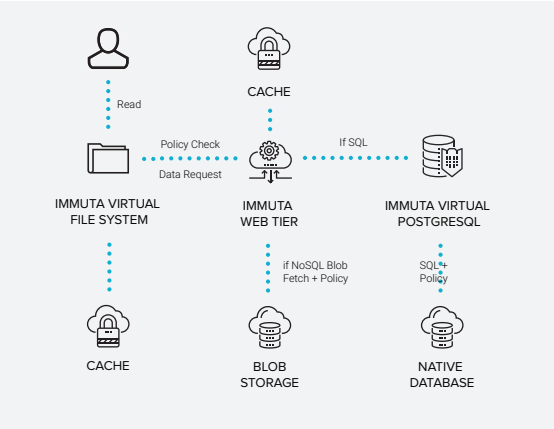
Filesystem: Users have the ability to mount an Immuta filesystem. This filesystem, like the SQL connection, is virtual. The files represent connected data from across your organization in a directory hierarchy, yet all the files are empty. Once a file is read, the file is hydrated dynamically with the data from the underlying storage technology with the policies enforced on-the-fly. Unlike the interactive SQL, the virtual filesystem does cache results removing load from the remote storage technology and reducing latency from the client. That cache's time to live is configurable per data source exposed in Immuta.

HDFS: Unlike the prior two access patterns, the Immuta HDFS access pattern is not virtual. The value in HDFS processing is to bring the code to the data, and as such, requires the Immuta policies to be enforced in-place on the data in the HDFS data nodes. Because of this, the Immuta HDFS layer can only act on data stored in HDFS. However, you are able to build complex subscription and access policies granularly on objects stored in HDFS and retain all the rich audit capabilities provided by the other Immuta virtual layers.

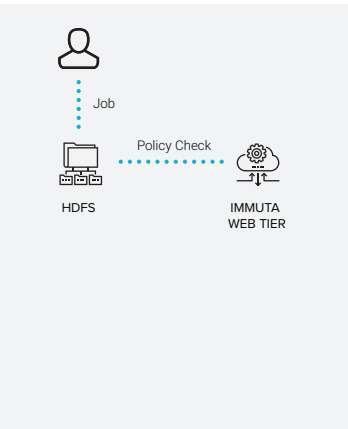
SQL



File System



HDFS



Integrations

Due to the fact that any tool or language can read either SQL, from a filesystem or HDFS, any upstream tool can communicate and integrate with the Immuta data control plane seamlessly and invisibly.

SQL

Anything that can "talk" SQL...




and more...

Filesystem

Anything that can read from a filesystem...




and more...

HDFS

Any batch processing with HDFS as its storage tier...



and more...

Beyond integrating with existing work invisibly in-flight, the Immuta access patterns also map nicely to data science workloads. The Filesystem is great for unstructured data processing as well as model training with libraries that expect to read data into memory, typically on small to medium sized data. The SQL layer is perfect for interactive data exploration, visualizations, and slicing/aggregating data. The SQL layer can act on any sized data, assuming the client isn't going to bring back "all" the data in scenarios when the data is large. The HDFS layer, considering it's not virtualized, is purpose-built for massive scale batch data processing, which can be helpful for aggregate or "needle in the haystack" analyses when all the data needs to be processed. It's also very handy for massive data transformation workloads.

An Access Pattern for Every Workload

FILE SYSTEM

- Model Training
- Unstructured Analysis
- ML Library Processing

Data Size: SMALL MEDIUM LARGE

SQL

- Interative Exploration
- Data Visualization
- Data Slicing & Aggregation

*Data Size: SMALL MEDIUM LARGE

HDFS

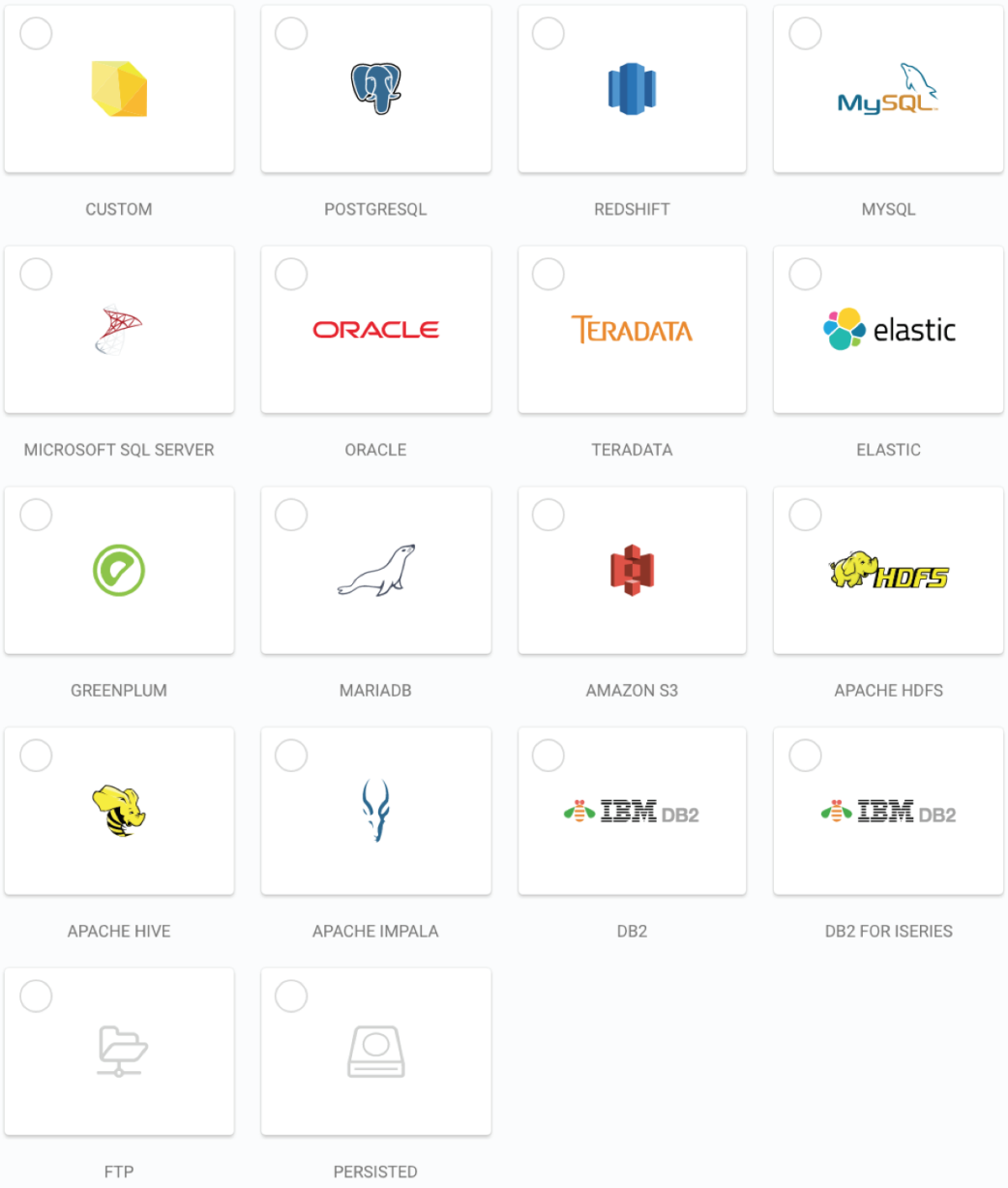
- Batch Processing of "All" your Data
- Needle in a haystack Analytics
- Transformations

Data Size: SMALL MEDIUM LARGE

* Assumes in "large" scenarios not all data is coming back to the client.

Also, since Immuta can present data in filesystem representations, it's the only virtual control plane on the planet that can expose NoSQL sources. Any storage technology, to include APIs, can be integrated as an upstream data source. Currently Immuta supports the following, but are always adding more based on customer demand:

Currently Supported Storage Technologies



Also note there is the option for custom data integrations, which can be done by interacting with the Immuta API. This includes any upstream or downstream feature of Immuta, in fact, the Immuta user interface can be completely ignored by integrating Immuta into existing platforms completely through our RESTful APIs.

Data Policies

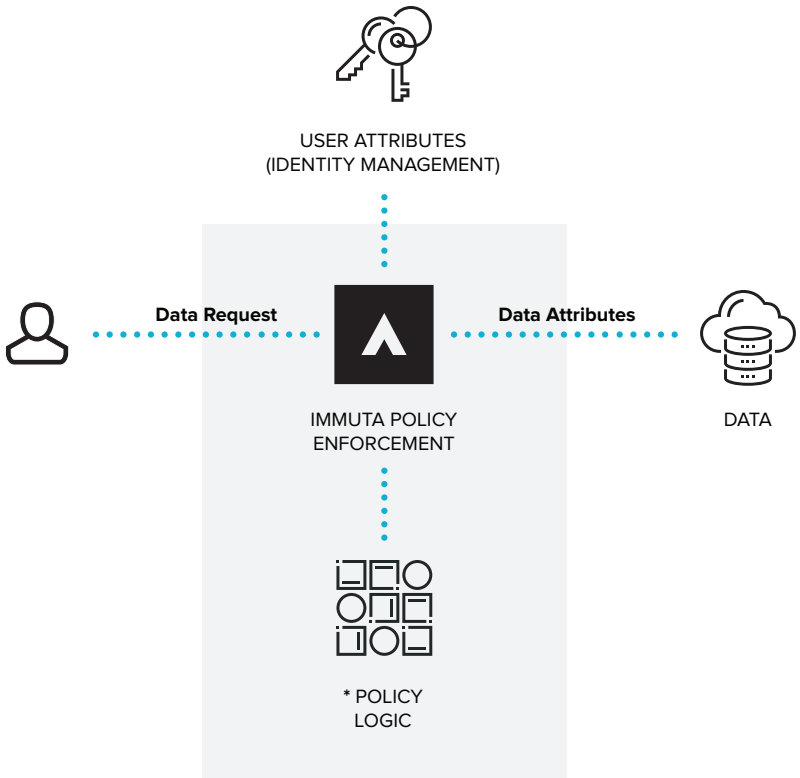
Immuta at its core is a policy enforcement point. To execute enforcement of policies on data, two to three inputs are required, depending on the type of policy.

First, and always required, are the user's attributes as provided by an external identity management system (IDAM). Immuta is completely pluggable to any IDAM and has built-in integrations with ActiveDirectory, LDAP, and Oauth. Immuta also has a self-contained IDAM if customers do not have one of their own.

Second, assuming you wish to enforce a policy, is the policy logic. The logic must include an action to take on the data as well as a condition that is built with attributes from the IDAM. For example Do Action WHEN Condition. Immuta has a clean and easy to use policy builder that does not require writing code - anyone can do it! However, should an organization already have a service that provides policy logic, that can be integrated into Immuta.

Lastly, some row level security policies need to have hooks into the data to make decisions on what to hide; those data attributes are matched against the policy logic to make decisions on if that row/object should be visible to the user. Data attributes are typically part of the data being exposed as a column or metadata attribute.

Immuta does incorporate configurable caching for all of the above components of a policy to improve performance.



* Typically in Immuta but can be external.

Immuta has two separate concepts of policy enforcement:

Subscription Policies: These are policies that determine if users can access the data at all. Subscriptions policies a configurable by the following:

- Anyone: Users will automatically be granted access at request time.
- Anyone Who Asks (and Is Approved): Users will need to request access and be granted permission by a data owner.
- Users with Specific Groups/Authorizations: This can avoid the manual approval process by creating condition(s) for which users can be granted access based on their attributes (from your Identity Management System).
- Individual Users You Select: You can hide the existence of the data altogether, and manually subscribe users you choose.

Data Privacy / Anonymization Policies: These are policies that are enforced on the raw data at query time, allowing you to "personalize" based on the conditional logic in the policy.

Masking Policies: You would use these to hide values in data. The various masking policies have various levels of utility while still preserving data privacy:

- Hashing: Hash the values to an irreversible hash, which is consistent within the data source so you can count or track the specific values (or even join with another table using this value), but not know what the actual value is.
- Constant: Make all the values in that column null or replace them with a constant, such as 'redacted'.
- Replace: Using a regular expression, replace certain parts of the value, for example convert a phone number (301) 555-9842 to (XXX) XXX-9842
- Rounding: Also termed k-anonymization; group values with their neighbors to prevent link attacks. For example, round precise dates to the nearest month, or bucket age to the nearest 10 year mark.

Row level security policies: These will hide entire rows or objects of data based on the policy being enforced, remembering some of these policies require the data to be tagged as well.

- Matching: Match a user attribute with a row/object/file attribute to determine if that row/object/file should be visible or not.
- Time Window: Restrict access to rows/objects/files that fall within the last x days/hours/minutes.
- Minimization: Restrict access to only a limited percentage of the data, randomly sampled per user.
- WHERE clause: (coming soon) Restrict access based on a WHERE clause that is executed based on the policy condition.

Differential Privacy: Immuta's patented technology to take the concept of differential privacy out of academia and put it in the hands of commercial customers. Differential privacy provides mathematical guarantees that you cannot pinpoint an individual (row) in the data. It is foolproof anonymization by understanding the sensitivity of the query, and using that, applying the appropriate noise (if any) to the response. For example "average age" could be changed from 50.5 to 55 at query time. To do this the Immuta SQL layer restricts queries run on the data to only aggregate queries (AVG, SUM, COUNT, etc) and prevents very sensitive queries from running at all, rather than adding a significant amount of noise.

Purpose-based policy: Manage access to data based on the purpose for which it's being accessed. This is very different from how organizations typically think about access control, yet critical to the future of privacy enforcement with regulations such as GDPR. The concept is simple, yet very difficult to achieve because historically we've always thought of user attributes as fairly static - you are not updating your IDAM several times a day. Yet, the context for which data is being analyzed is constantly changing, and with regulations like GDPR, data can only be processed for the purpose for which it was consented by the data subject. This requires constant change where a new user attribute is required - purpose. Immuta supports purpose through "projects" where a project has a purpose, and Immuta provides a data access point through that project to interact with the data under that purpose - and audit it as such. You can use purpose to build your policy logic conditions just like attributes from your IDAM.

Lifecycle of a policy-enforced query in Immuta

The following actions occur on Immuta data fetches (non-HDFS) to enforce the dynamic/complex policy logic with some potential caching occurring on steps 1 and 2:

1. Interacts with the enterprise identity management system to understand the groups/authorizations the authenticated user possesses.

2. Fetches the policies currently enforced on the data source being queried.

3. Blends the fetched policy with the user attribution to build the literal policy logic.

4. Executes the query on the native data source including any policy logic that can be enforced as part of the query execution, e.g. pushed down to the native database for processing.

5. Response of the query passes through Immuta, enforcing any remaining policies, and back to the requesting client.

6. Audits the query.

For HDFS only steps 1 through 3 and 6 occur, but at job startup time, and the data reads happen on the HDFS data nodes with the policies being enforced based on the data returned from step 3. Auditing does occur on those reads.

Immuta Deployment

Immuta's server-side software is comprised of four major components.

- **The Immuta web service**. This service is responsible for all web-based user interaction with Immuta, metadata ingest, and is the service that backs the Immuta virtual file system.

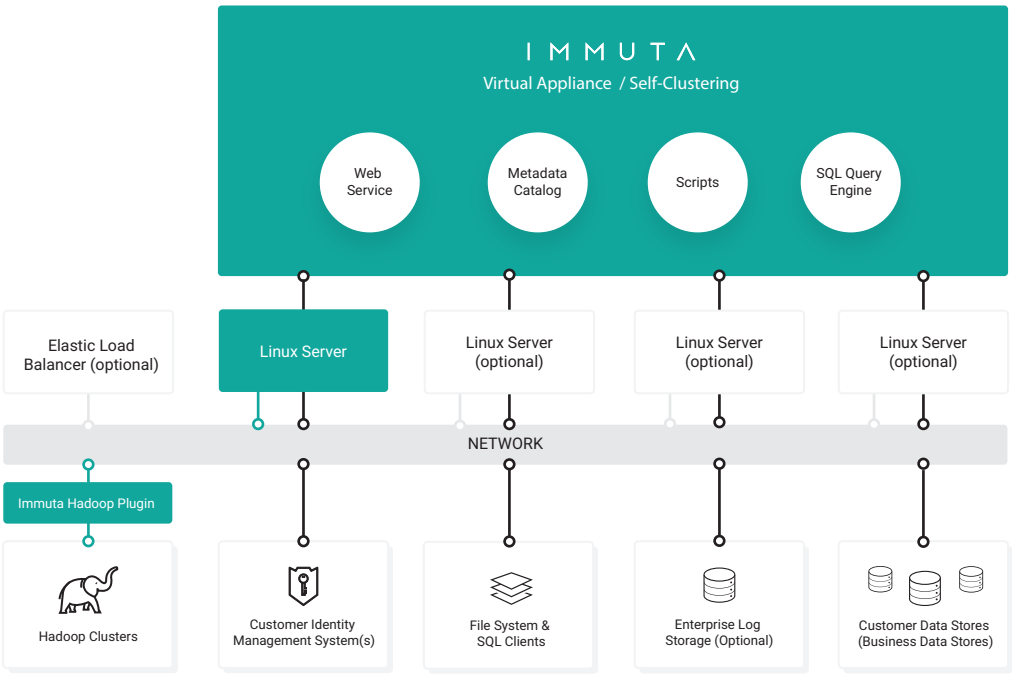
- **The Immuta metadata catalog**. This internal database maintains a small amount of data on each object you register with Immuta so that Immuta can provide responsive access to objects to your users while enabling you to dynamically create access policies on the objects.

- **The Immuta SQL query engine**. This service tracks queryable data source configuration and exposes a SQL connection to the Immuta web service and to any SQL client such as a SQL library in Python or R or a BI / Data Science tool. This service interprets client SQL queries, pushes queries to your connected business databases, applies policies, and returns query responses to your SQL clients.

- **Immuta HDFS layer (OPTIONAL)**. The Immuta HDFS layer allows Immuta to enforce data access policies within your Hadoop infrastructure. To enable this functionality, Immuta supplies two .jar files to be installed on your name node(s) and data nodes.

- **The Immuta Scripts environment (OPTIONAL)**. This optional service provides a hosted Jupyter notebook environment to your analysts and data scientists. It allows them to share scripts and provides them instant access to data via pre-configured virtual file systems and sql connections.

Immuta's standard installation requires minimal administrative interaction. Immuta can run on a single RHEL/CentOS 7 machine or on a cluster of such machines. Cluster management is built into Immuta and administering an Immuta cluster is much more like administering a virtual appliance than managing a distributed system. Additionally, the standard cluster installation is preconfigured with high availability, scalability, and resource scheduling. For full technical details on the standard installation and for other installation types please review our Immuta Install Guide.



Scalability

Immuta is designed to be scalable in several dimensions and, for the standard Immuta deployment minimal administrative effort is required to manage scaling beyond the addition of nodes to the Immuta system. Scalability can also be achieved in non-standard deployments, but requires the time of skilled systems administrator resources.

- The Immuta web service is responsible for all web-based interaction with Immuta, metadata ingest, and is the service that backs the Immuta virtual file system. This service is stateless and horizontally scalable.

- By keeping a metadata catalog rather than maintaining separate copies of data, Immuta's database is designed to remain small and responsive. By running replicated instances of this internal database, the catalog can scale in support of the web service.

- The Immuta SQL Query Engine which backs Immuta's SQL Access Pattern can scale horizontally with user load. Individual queries are limited by the memory allocated to an individual instance in scenarios where queries can not be fully pushed-down to business databases.

- If using the optional Immuta Script environment, each Script instance runs in its own Docker container. Thus, the limits of an individual script are tied to the RAM/CPU available on individual hosts or the limits of your Spark cluster when using Spark. The number of concurrent scripts can scale horizontally with the infrastructure backing your Immuta system.

High Availability

Because each component of Immuta is designed to be horizontally scalable, Immuta can be configured for high availability. Upgrades and major configuration changes may require scheduled downtime, but even a failure of Immuta's master internal database, recovery happens within seconds. With the addition of an external load balancer Immuta's standard deployment comes preconfigured with these availability features.

Security

Immuta’s core function of policy enforcement and management is designed to improve your data security. Beyond this primary feature, Immuta protects your data in several other ways.

First, Immuta is designed to leverage your existing identity management system when desired. This allows Immuta to benefit from the work your security team has already done to validate users, protect credentials, and define roles and attributes.

Next, by default, all network communications with Immuta and within Immuta are encrypted via TLS. This ensures your data is protected while in transit.

By design, Immuta does not make any persistent copies of data. The Immuta file system allows for the temporary local caching of file representations of data for performance purposes, but all data is encrypted on disk, the encryption keys are never written to disk, data is only decrypted on read by the Immuta virtual file system, and user access to data is revalidated each time the Immuta virtual file system opens a virtual file.

Benchmarks

Immuta provides a scalable, highly available, data abstraction layer that allows complex policy enforcement on-the-fly rather than batch pre-computing various different anonymized data sets - because nobody has static data nor static policies. The benefits of this are the ability to maintain policies in a single location across all your data silos, change those policies at will, have those changes reflected immediately, and have consistent audit logs across your data silos. This allows the execution of many more complex policies as each policy doesn't result in a new full copy of the data to match every user/policy combination.

This power and flexibility requires some computation to occur at the Immuta layer between the data being queried in the native storage technology and the analyst wanting to analyze the data. While some overhead is to be expected, Immuta provides a solution that minimizes this tradeoff significantly - providing you with the flexibility of dynamic policies. When considering the impact of Immuta queries, you should compare it against not only the query response times, as we will demonstrate below, but also the business process "red tape." We've seen months of manual approval processes and data anonymization translations before data is in the hands of the downstream user due to policies and regulation - which Immuta eliminates.

Constant Immuta Overhead

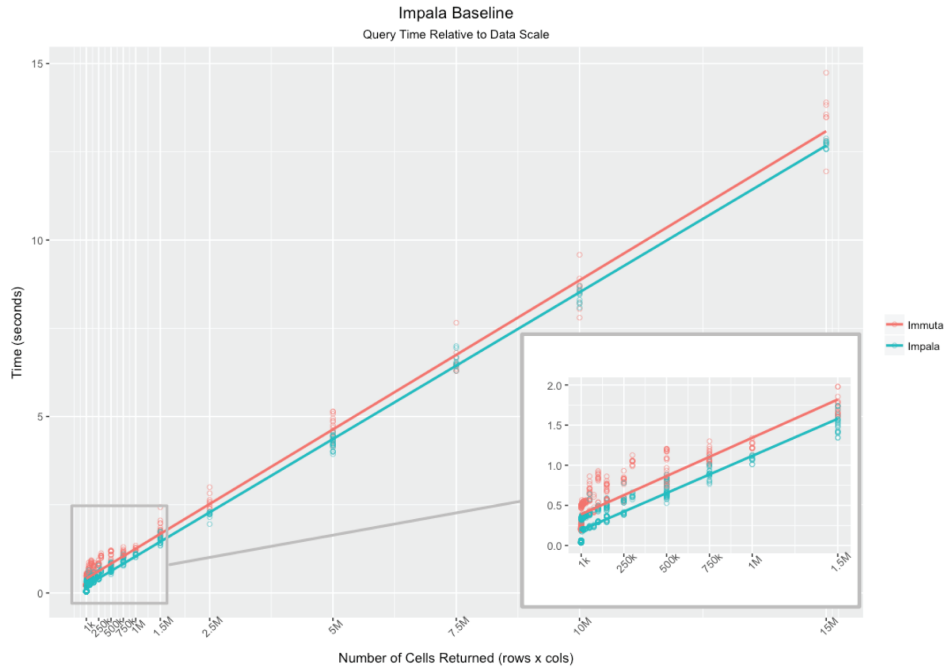
Without any policies at all, Immuta introduces a constant overhead due to steps 1, 2, 4 and 5 described in the "Lifecycle of a policy-enforced query in Immuta" section above. That constant overhead, in a recommended Immuta deployment configuration, is ¼ of a second, no matter the query.

Additionally, due to step 5 in the "Lifecycle of a policy-enforced query in Immuta" section above, the latency increase scales linearly with returned data volume and that amount varies slightly based on the database type. Image 1 shows our performance test results while querying an Impala datasource with no policies. Although the lines look parallel, there is a slight slope difference showing that Immuta's overhead increases with data scale. At 15 million data points, a half second of overhead is added as compared to querying Impala directly, so a 12 second direct query will take 12.5 seconds through Immuta. Extrapolating these curves shows that Immuta will add an additional second to a query that returns 75 million data points.

Constant Immuta Overhead

Without any policies at all, Immuta introduces a constant overhead due to steps 1, 2, 4 and 5 described in the "Lifecycle of a policy-enforced query in Immuta" section above. That constant overhead, in a recommended Immuta deployment configuration, is ¼ of a second, no matter the query.

Additionally, due to step 5 in the "Lifecycle of a policy-enforced query in Immuta" section above, the latency increase scales linearly with returned data volume and that amount varies slightly based on the database type. Image 1 shows our performance test results while querying an Impala datasource with no policies. Although the lines look parallel, there is a slight slope difference showing that Immuta's overhead increases with data scale. At 15 million data points, a half second of overhead is added as compared to querying Impala directly, so a 12 second direct query will take 12.5 seconds through Immuta. Extrapolating these curves shows that Immuta will add an additional second to a query that returns 75 million data points.



Impala Configuration:

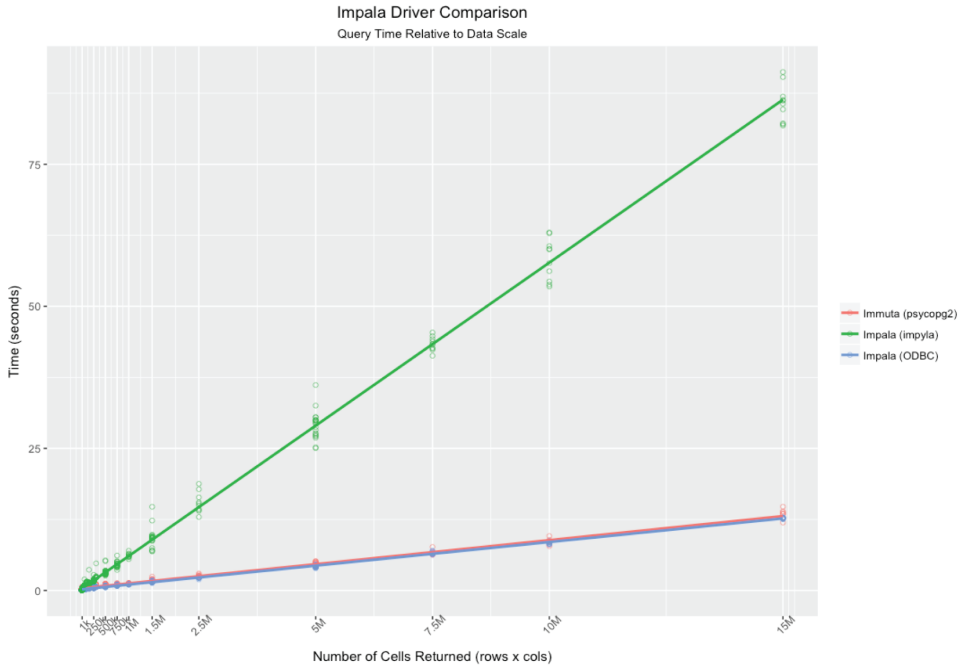
- CDH 5.7.6 HIVE
- 10 Nodes, 9 Datanodes (8 cores, 32 GB RAM)
- ~350 GB NYC Taxi Data (170 million rows, 180 columns)

Immuta Configuration:

- 5 Node install of Immuta v. 1.5
- 1 @ 4 vCPU, 16 GiB RAM, 64 GiB SSD
- 4 @ 16 vCPU, 122 GiB RAM, 256 GiB SSD

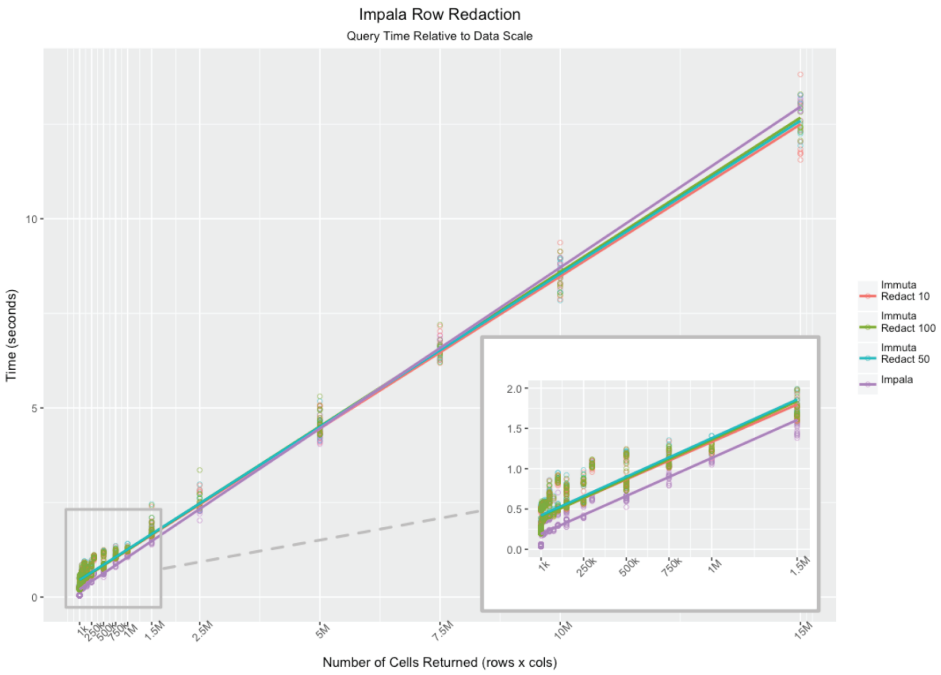
SQL queries were executed 10 times for each: directly to Impala, and via Immuta to Impala, taking the average response times.

Note that choosing the right database driver could have a more significant impact on the query response times than Immuta. As shown in Image 2, using the impyla library to connect directly to Impala (no Immuta at all) showed query times increasing dramatically as data scale increased.



Impact of Row Redaction

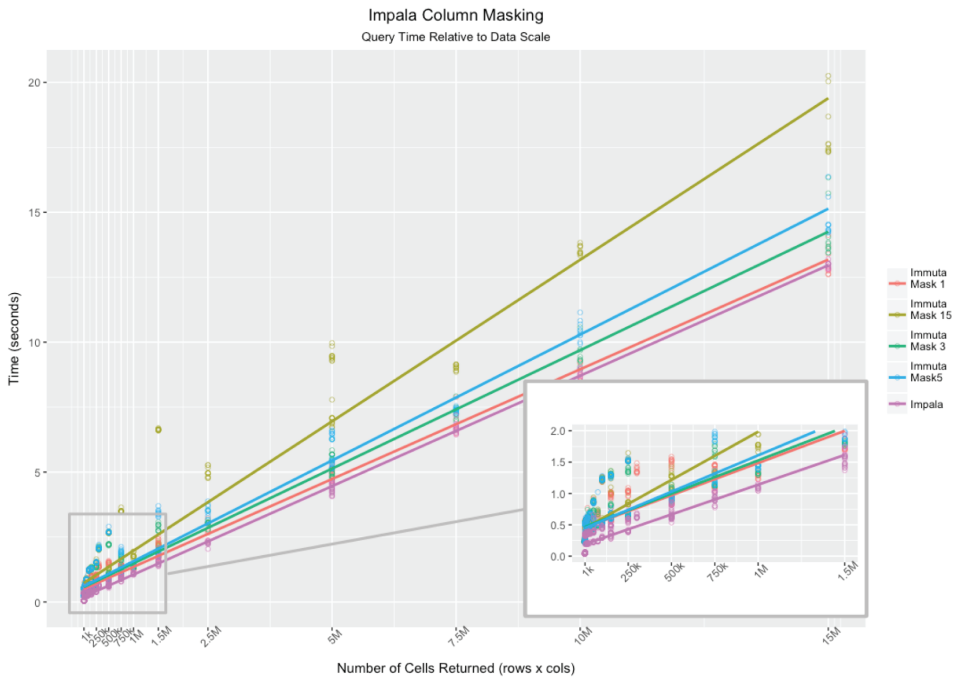
Row redaction has very comparable overhead to the constant Immuta overhead discussed above because the redaction policies are always pushed down to the native database as part of the where clause. The below image depicts the impact of row redaction on Immuta queries, where the percentage of the data protected (or returned, depending how you look at it) has an impact simply because the where clause differences on the pushed-down policy.



Impact of Column Masking

Column masking is sometimes enforced as the data is streaming through Immuta, so there can be an impact on performance based on how many columns are being masked. The only column masking policies that must be enforced while the response is streaming through Immuta are hashing and regex. Both rounding and replace with constant/null are pushed down to the remote data source and thus have no impact on performance beyond what was discussed above.

Below are charts depicting the impact of column masking (using hashing, e.g. not pushed down), focusing on the amount of masking policies being enforced. As you can see, as you add more masked columns, performance will degrade. You should consider carefully what kind of masking policies should be enforce and what columns you should choose when exposing data in Immuta.



The filesystem has identical overhead with the interactive SQL as both are going through the same process as depicted in the access pattern diagram above. However, the filesystem does cache the response as files on disk and in the Immuta caching layer. Subsequent reads of those files will be read directly from disk and will be substantially faster than querying the remote storage technology.

IMMUTA

About Immuta

Immuta is the fastest way for algorithm-driven enterprises to accelerate the development and control of machine learning and advanced analytics. The company's hyperscale data management platform provides data scientists with rapid, personalized data access to dramatically improve the creation, deployment and auditability of machine learning and AI.

immuta.com

1-800-655-0982 or 1-240-473-2731

Immuta, Inc. 8400 Baltimore Ave, Suite 100, College Park, MD 20740