eBOOK

The Layers of Trust

Immuta delivers the intersection of privacy and security to de-risk cloud analytics for infosec teams while preserving utility for data analytics.

IMMUTA™

INTRODUCTION

Security is a driving force, particularly for organizations making the move from on-premises databases to SaaS-based data warehouses where users outside the organization (cloud provider and SaaS provider) need to be trusted at some level.

This massive shift to the cloud – with its security concerns – has been coupled with a rapid increase in privacy issues. To quote a recent Cisco report: "Over the past few years, data privacy has evolved from "nice to have" to a business imperative and critical boardroom issue." Security and privacy are colliding in the cloud and creating challenges organizations have never seen. To tackle this problem, it's critical to think about layers of trust and where trust lives in conjunction with data utility tradeoffs.

To set the stage, let's first focus on privacy. There are really four data categories of privacy:

Direct Identifiers

Think credit card number, full name, drivers license

Indirect Identifiers

Your sparsely populated zip code, your rare car make/model, a rare disease you have

Sensitive Information

Your sexual preference, a disease you have

Other data

As you can see, sensitive information can overlap with indirect identifiers (e.g., a disease you have). This makes indirect identifiers interesting; they can straddle the line between both sensitive information and direct identifiers. How can an indirect identifier be a direct identifier? It's not very hard, and it has been the source of many privacy attacks, such as the Netflix Challenge privacy scandal. When Netflix released their data for the challenge, it contained no direct identifiers indicating who rated what movie. It simply included the ratings and other information to help build movie predictions – one would think this is simply categorized as "other data." However, by taking the ratings and comparing them to other rating sites, an attacker was able to identify the movie raters which led to the lawsuit (this is termed a link attack). The ratings themselves – presumably the most important part for the movie prediction algorithms – is in fact an indirect identifier!

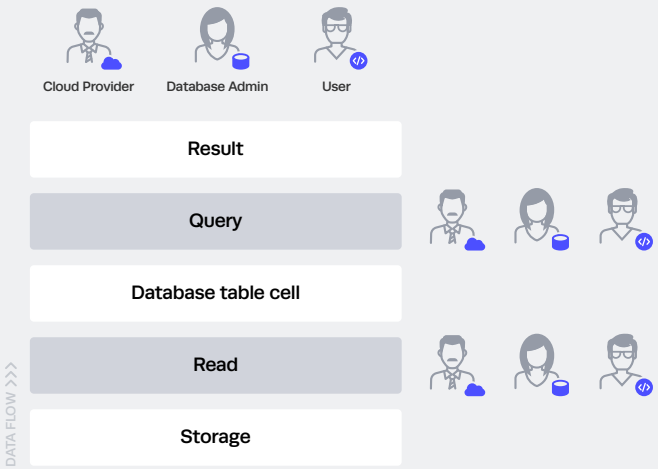
To effectively manage privacy, you not only have to mask direct identifiers, but, you also need to mask indirect identifiers and sensitive information – and here's the key part – *from your own employees*. This creates a conundrum, because the indirect identifiers and sensitive information in your data are exactly the same data you want/need to analyze, as we saw from the Netflix Challenge. In other words, you can't blindly mask for privacy without losing utility, and you can't provide utility from the data without losing privacy. So what do you do? We'll come back to that.

Security is a different but related beast.

If someone breaks into your database account, they can immediately read your data and see everyone's credit card number, for example. Or worse, your own disgruntled database administrator could be the culprit. If someone has unfettered access to your data, they can see all your data no matter the privacy category.

Consider the following diagram and table and imagine there are no controls at all:

Everyone Trusted

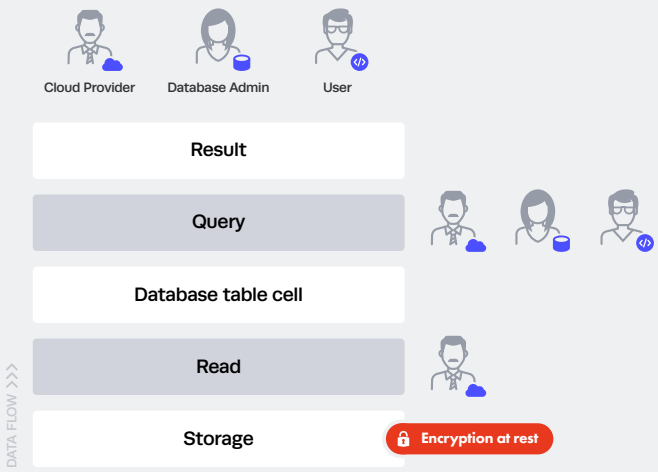


last_name	gender	race	ssn
Baskerfield	Female	Samoan	178-91-3640
Korba	Female	Ute	118-47-0661
Dobell	Male	Potawatomi	306-41-1756
Pittman	Female	Melanesian	490-30-3360
Lydiate	Female	Micronesian	842-48-8198
Shrive	Female	Spaniard	190-17-7902
Rymer	Male	Venezuelan	796-78-0964
Toretta	Female	Argentinian	543-51-7257
Chainey	Female	Paraguayan	629-41-1710
Bayles	Female	Cherokee	809-01-9630

In this scenario, all three personas can run queries against any table. Additionally, they could bypass the database altogether and read directly from storage (note that we are oversimplifying "read" from storage in this example; this does not mean through a service, but literally reading the data directly from the storage system).

This architecture is undesirable for any organization and defeats the purpose of the database and database controls (which we will revisit later).

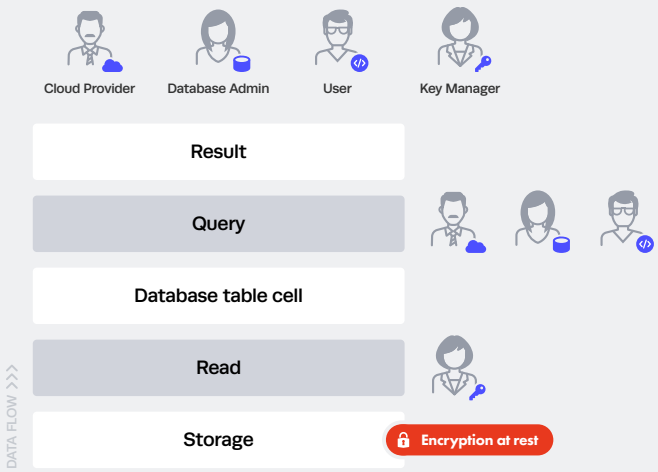
Now consider a scenario in which encryption at rest from a cloud provider is being used:



last_name	gender	race	ssn
Baskerfield	Female	Samoan	178-91-3640
Korba	Female	Ute	118-47-0661
Dobell	Male	Potawatomi	306-41-1756
Pittman	Female	Melanesian	490-30-3360
Lydiate	Female	Micronesian	842-48-8198
Shrive	Female	Spaniard	190-17-7902
Rymer	Male	Venezuelan	796-78-0964
Toretta	Female	Argentinian	543-51-7257
Chainey	Female	Paraguayan	629-41-1710
Bayles	Female	Cherokee	809-01-9630

Using encryption at rest, only the cloud provider can decrypt the data on disk in storage because they possess the decryption key. But this invites the question: how do the database queries work? In the read interface between storage and the database, data is "in the clear" as far as the database is concerned. This is how storage on a laptop is encrypted, it is not apparent that every time a file is opened that it is actually being decrypted for the user in that read interface. This is the same reason why everything is still visible in the table, it has a similar read interface decrypting between storage and the database. So what is this protecting against? Not much, to be honest. If someone stole your cloud provider's hardware, they would only be able to see encrypted data on the disk instead of in the clear.

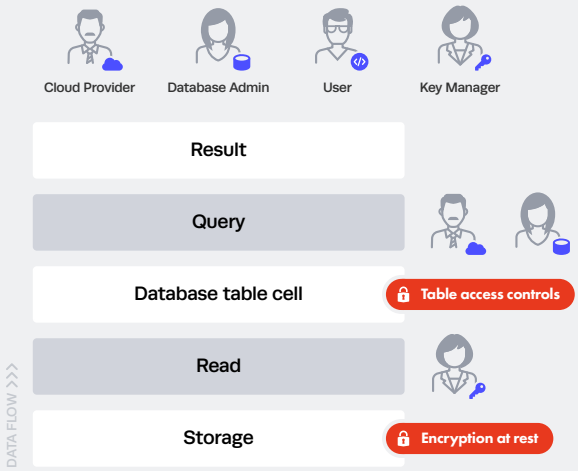
To address this, you can control your own encryption keys. This way, although the cloud provider still enables that read interface to make everything in the clear, you can provide the key so the cloud provider can't decrypt on their own. This is still not a full solution, however, as we are introducing yet another persona – the key manager. By controlling your own encryption keys, you are just shifting trust elsewhere. But this isn't necessarily a bad thing in this scenario, as you probably trust this person more, the keys can live outside the cloud, and the key manager can only access the key management system – not the database.



last_name	gender	race	ssn
Baskerfield	Female	Samoan	178-91-3640
Korba	Female	Ute	118-47-0661
Dobell	Male	Potawatomi	306-41-1756
Pittman	Female	Melanesian	490-30-3360
Lydiate	Female	Micronesian	842-48-8198
Shrive	Female	Spaniard	190-17-7902
Rymer	Male	Venezuelan	796-78-0964
Toretta	Female	Argentinian	543-51-7257
Chainey	Female	Paraguayan	629-41-1710
Bayles	Female	Cherokee	809-01-9630

This scenario obviously doesn't change what is seen in the table at all because we've only modified how the storage decryption key is managed. And that's as far as we can go on storage because it just results in shifting trust elsewhere.

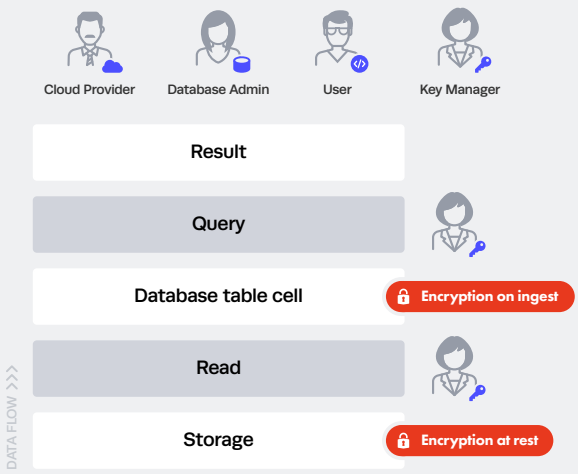
Now here comes the fun part. You don't want everyone to be able to query every table, so to ensure this won't happen we now have our database administrator (DBA) implement table level controls, thereby restricting who can query which tables. It results in this picture:



last_name	gender	race	ssn
Baskerfield	Female	Samoan	178-91-3640
Korba	Female	Ute	118-47-0661
Dobell	Male	Potawatomi	306-41-1756
Pittman	Female	Melanesian	490-30-3360
Lydiate	Female	Micronesian	842-48-8198
Shrive	Female	Spaniard	190-17-7902
Rymer	Male	Venezuelan	796-78-0964
Toretta	Female	Argentinian	543-51-7257
Chainey	Female	Paraguayan	629-41-1710
Bayles	Female	Cherokee	809-01-9630

Now your users can't go around querying any table they want. All of the table above is still in the clear, but only to certain people. For example, your DBA would be able to see it, along with the users who were entitled to it. Theoretically, your cloud database provider would also be able to see it because they would know how to get around the controls they provided you – this is doubtful and bad for business, but worth mentioning.

So we're still fairly exposed and this is where things start to get tricky, which brings us back to our different categories of privacy. A common idea is: Let's encrypt all the data in the database like we did in storage and we'll control the key! Encrypting (or tokenizing) the data before it lands in the database can be commonly termed *encrypted on-ingest*, (i.e., you encrypted the data before it ever landed in storage). This means that the data is encrypted twice in storage, on the way in, and again at rest. Because of that, when it's decrypted out of storage through that interface, it's still encrypted because you did it on ingest. If you take this approach, now only your key manager needs to be trusted.



last_name	gender	race	ssn
T0RobVpqY3pNMlkzTldVNFpEV...	T0RVMVkyUXdZbUUzWkRKalp...	TVdJME9EVXpPV1EyT0RVMk9...	prMjCMmSo5g=:4vivaj7tmpGk+hjC0W0gaQ==
TXpWbVpHRmtZemxpTnpZMk5...	T0RVMVkyUXdZbUUzWkRKalp...	WTJJNFptRmhNMlpsWXpjeFky...	VFK6lvJ1d68=:dz8wtuV0njXpFZlKmo95jg==
T1dOaU9UVXhNMkUwWWpRMV...	WW1FMU5ETTVNekUwTTJFeF...	Wm1Sa1pXTTRaV1ZtTkRFNFI6...	Mjc5GAAzULU=:4wkjKQI3WNrUEVWZexXtCA==
TXpFMFltVmtPRFJoTWpnNVpq...	T0RVMVkyUXdZbUUzWkRKalp...	TURJNE9XVmlabVI6Tm1Oa1IU...	TzzEyS2P2LA=:Jkid3z98cgcTik/TsKY2Pw==
TURjNE9XWmhOR0UwWm1VMK...	T0RVMVkyUXdZbUUzWkRKalp...	WW1VeFpESTRNMkV3TURSa1...	V3M7g+5GJJM=:OvRR8maUE4Dj+gW//o69XQ==
TXpnd01EWmpNbVpoWTJVMlp...	T0RVMVkyUXdZbUUzWkRKalp...	TkRCbU1tSTBPC0pqTm1JeVlU...	5EqSayRGKWs=:LNzY7gsCEr67i+3+e6Eh6Q==
TURSbU1XSTJObU00TURKalIqW...	WW1FMU5ETTVNekUwTTJFeF...	TmpReU56TmhNMlU0WTJGak1...	5zM5YlSGHZc=:l/moeZ7fXWdo4cmr9HJdew==
WVdSall6a3dZelpsTkdSa1pXSm...	T0RVMVkyUXdZbUUzWkRKalp...	WmpNM1pqTTBNREU1TVdSa0...	zAMBvSAqERo=:1TtOMmDPBJvTw3cBBDqv7A==
WlRoa09HVXdPVFk0TkdObFlqW...	T0RVMVkyUXdZbUUzWkRKalp...	WVdWalpXUTVNVGxpWmpJM0...	cleHnbAng44=:TUQsyC1Hh+E7ssWyY9LZUQ==
WkRnek5tUXpaRGN5TXpjek1Ea...	T0RVMVkyUXdZbUUzWkRKalp...	WmprMFpUTmhNR1ZoWWpaaE...	8DM8JxM9r7U=:0ezhZPGqcWUkIM2GVX6sGw==

This is actually a terrible idea because it means your database is now completely worthless.

Why?

Let's look at an even simpler example:

Because everything is indexed by the encrypted value (as you can see in the table above), you just turned this row in your database:

```
name: Steve
credit_card_number: 123456789
age: 44
address: 8787 Diamondback Drive, College Park MD
```

into this:

```
name: 3098jgp9iwr9iewgp9rweifg9e
credit_card_number: 09q3jepf3q0e8fefg8
age: 09q34ejtgpe9rgpiofjopssd
address: 09q34peigpe9irfdinsifvods
```

Now when you run a query, you won't get any results because your database doesn't have the same fancy interface that existed between your storage layer.

```
SELECT * FROM table_x WHERE name = 'Steve'
```

...nothing...

```
SELECT * FROM table_x WHERE age > 40
```

...breaks. You just tried to query a text column with a numeric operation, this is because the column had to be converted to text to encrypt the values it contains.

But what if we tokenize the data?

It's the same problem. With tokenization, you are merely replacing the garbled, encrypted values with just as useless, but more visually pleasing, fake values that also can't be queried in a meaningful way. This only addresses the "column type" problem.

Instead, what if we added the fancy interface that knows how to decrypt on the fly, like we have with the storage layer!? That certainly can be done, but it only solves a small piece of the problem. This is because the read from storage is "dumb." It just needs to read blobs of data and have the interface decrypt. While the interface between the database and the data must be "smart" (i.e., it needs to only pull relevant data from its index based on a query), all the indexes are still based on the encrypted values. To make this work:

```
SELECT * FROM table_x WHERE name = 'Steve'
```

The interface must convert the Steve into the encrypted value and push that down to the database so it can find the encrypted version of Steve, like this:

```
SELECT * FROM table_x WHERE name = '3098jgp9iwr9iewgp9rweifg9e'
```

Voila! You get results, and they can be decrypted on read:

```
name: Steve
credit_card_number: 123456789
age: 44
address: 8787 Diamondback Drive, College Park MD
```

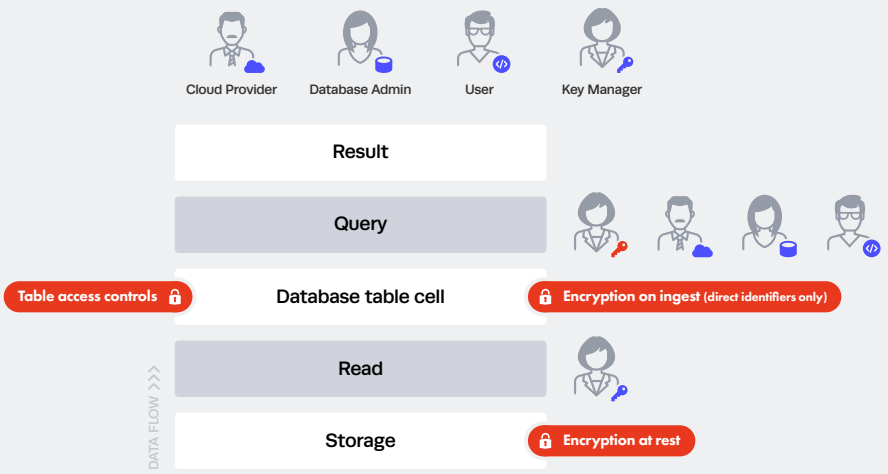
This is still problematic as it only works well for equality ("=") queries.

What about this query?:

```
SELECT * FROM table_x WHERE age > 40
```

Because the encryption does not take order into account, this isn't going to work at all. For example, 40 could get encrypted to xyz and 41 could get encrypted to abc. Tokenization will have the same issue. There are order preserving encryption algorithms, and other encryption algorithms that can support fuzzy search, but they are less secure encryption algorithms. (To learn more about them, take a look at this paper from MIT CSAIL, CryptDB: Protecting Confidentiality with Encrypted Query Processing.) The bottom line is, if you were to encrypt all your data in a way that also makes it useful, you'd lose a lot of the guarantees due to the weaker encryption algorithms.

Where does this leave us? You should encrypt on ingest sparingly or not at all. If done, it should be done against your most risky data, typically direct identifiers – essentially anything that must be hidden at all cost from your DBA and cloud provider. This also happens to be the category of data with the least utility. Users won't query it often (and if they are, it will be equality queries) or use it for their analysis. Real analysis is reserved for the juicy categories, like indirect identifiers and sensitive data. Here's our latest picture:



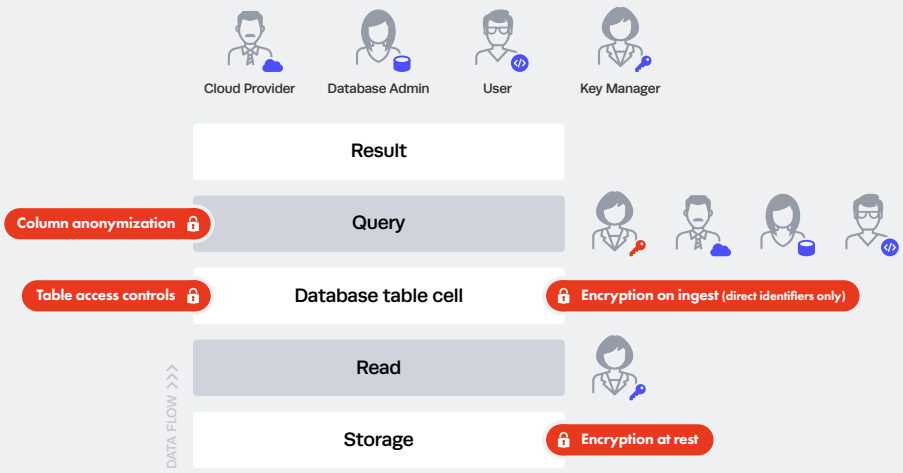
last_name	gender	race	ssn
Baskerfield	Female	Samoan	prMjCMmSo5g=:4vivaj7tmpGk+hjC0W0gaQ==
Korba	Female	Ute	VFK6lvJ1d68=:dz8wtuV0njXpFZIKmo95jg==
Dobell	Male	Potawatomi	Mjc5GAAzULU=:4wkjKQI3WNrUEVWZexXtCA==
Pittman	Female	Melanesian	TzzEyS2P2LA=:Jkid3z98cgcTik/TsKY2Pw==
Lydiate	Female	Micronesian	V3M7g+5GJJM=:OvRR8maUE4Dj+gW//o69XQ==
Shrive	Female	Spaniard	5EqSayRGKWs=:LNzY7gsCEr67i+3+e6Eh6Q==
Rymer	Male	Venezuelan	5zM5YISGHZc=:I/moeZ7fXWdo4cmr9HJdew==
Toretta	Female	Argentinian	zAMBvSAqERo=:1TtOMmDPBJvTw3cBBDqv7A==
Chainey	Female	Paraguayan	cleHnbAng44=:TUQsyC1Hh+E7ssWyY9LZUQ==
Bayles	Female	Cherokee	8DM8JxM9r7U=:0ezhZPGqcWUkIM2GVX6sGw==

As you can see here, everyone can see some of the data – the indirect identifiers and sensitive data – and we still have the table controls the DBA put in place. But only the Key Manager can see all of the data – in this case, the SSN column. This is good because neither your DBA nor your cloud database provider can see the highly sensitive direct identifiers (SSN) because they are encrypted using your key on-ingest. Also, if a data breach occurs, the attacker will only see the encrypted SSN. This is still not perfect because someone could breach your key management system and see the data. It all depends on where the breach occurs in your trust layers because all you did was push the trust somewhere else.

At this point, you still have indirect identifiers that are completely visible to users (which is not good). Now you've come full circle to privacy controls. Many times, a solution to privacy controls is to create multiple versions of the same table where different columns are missing so you can share those through your table controls as needed. But this isn't good enough. If you think back to the Netflix example, the movie ratings were both the privacy attack failure point and the primary data being used for the analysis. This cannot be a yes/no, binary control.

This is where anonymization techniques can be applied. These are algorithms you can use to "fuzz" the data in a way where it retains some level of utility yet also provides a level of privacy. One such technique is k-anonymization, which ensures there is no indirect identifier value revealed to the analyst/user which is unique enough to open the direct identifier(s) to a linkage attack. These techniques can be applied through data copies as well, just like we discussed with hiding columns. But why bother doing that? We can use our fancy interface trick where instead of decrypting-on-the-fly, we are fuzzing-on-the-fly.

This is so powerful because, in many cases, it allows you to query the underlying data in meaningful ways (which you can't do when the data is simply encrypted) and also utilize the results for your analysis. You are getting value from the data, while at the same time maintaining privacy. These techniques can be applied at query time to both indirect identifiers and sensitive data. Here's that picture:



last_name	gender	race	ssn
Baskerfield	Female	Samoan	prMjCMmSo5g=:4vivaj7tmpGk+hjC0W0gaQ==
Korba	Female	Ute	VFK6lvJ1d68=:dz8wtuV0njXpFZIKmo95jg==
Dobell	Male	Potawatomi	Mjc5GAAzULU=:4wkjKQI3WNrUEVWZexXtCA==
Pittman	Female	Melanesian	TzzEyS2P2LA=:Jkid3z98cgcTik/TsKY2Pw==
Lydiate	Female	Micronesian	V3M7g+5GJJM=:OvRR8maUE4Dj+gW//o69XQ==
Shrive	NULL	NULL	5EqSayRGKWs=:LNzY7gsCEr67i+3+e6Eh6Q==
Rymer	Male	NULL	5zM5YISGHZc=:I/moeZ7fXWdo4cmr9HJdew==
Toretta	Female	Argentinian	zAMBvSAqERo=:1TtOMmDPBJvTw3cBBDqv7A==
Chainey	NULL	NULL	cleHnbAng44=:TUQsyC1Hh+E7ssWyY9LZUQ==
Bayles	Female	Cherokee	8DM8JxM9r7U=:0ezhZPGqcWUkIM2GVX6sGw==

Look closely at the gender and race columns. We've applied the k-anonymization policy to them, which suppressed highly unique values (either in combination or by themselves) that could lead to a privacy breach with NULLs. However, we left the other values in the clear, providing a high level of utility from these columns rather than completely removing them. Immuta offers a wide variety of privacy enhancing technologies (PETs) like k-anonymization which allow customers to make tradeoffs between privacy and utility.

Why do it at query time? As the number of different privacy lenses into your data increases, applying these techniques as data copies results in ELT spaghetti and role explosion. It must be dynamic through the interface. Immuta provides an attribute-based access control (ABAC) model rather than a role-based access control model (RBAC). The key difference with ABAC is that complex, and completely separate, rules can evaluate many different user attributes rather than conflating WHO and WHAT they have access to in a role. As described by the NIST Guide to Attribute Based Access Control (ABAC) Definition and Considerations: at access-request-time, "the ABAC engine can make an access control decision based on the assigned attributes of the requester, the assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions. Under this arrangement policies can be created and managed without direct reference to potentially numerous users and objects, and users and objects can be provisioned without reference to policy."

ABAC allows for flexibility and scalability when building and enforcing policies, thereby helping you avoid the "role explosion" required to cover all column access and anonymization scenarios when using RBAC. The bottom line is that if you are concerned with fine-grained, column-level controls and anonymization, the RBAC model will not scale.

And there you have it...

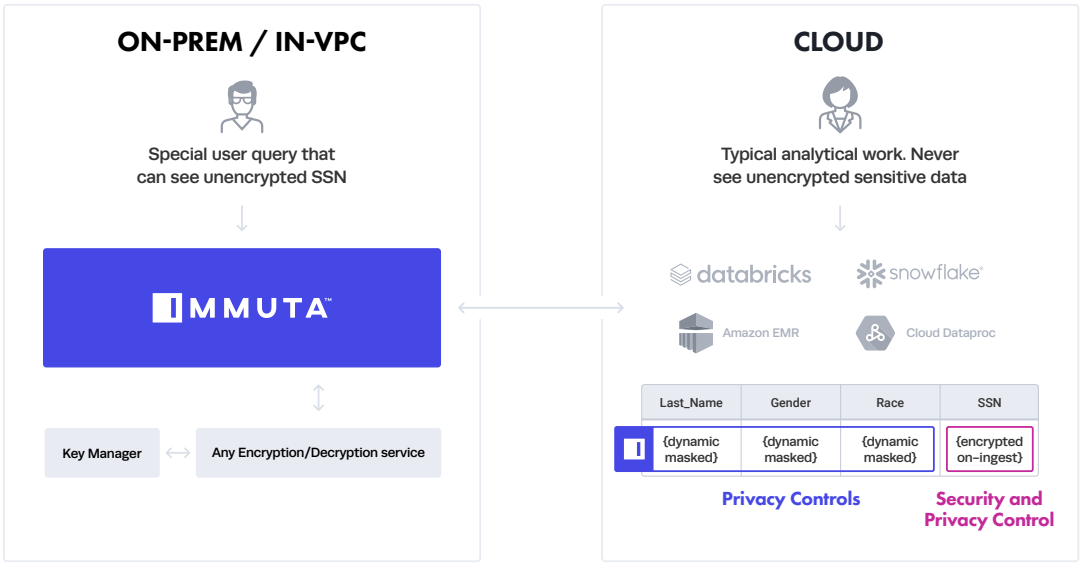
Security is maintained by managing your own encryption keys for the data encrypted-at-rest (storage) as well as for the highly sensitive encrypted-on-ingest direct identifiers in your database, if desired.

Privacy is maintained through dynamic fuzzing-on-the-fly, commonly termed dynamic masking, that occurs as part of the query interface to provide a level of utility with privacy. That same interface can manage the encryption/decryption operations on the direct identifiers that were encrypted on-ingest.

And yes, it is possible to not encrypt anything on-ingest and instead manage all controls in the dynamic masking interface. To do this, there are algorithms that completely obfuscate values rather than fuzz them. This provides much more flexibility and functionality, but it's important to remember that with this approach, some extra trust lies with the cloud provider and your DBA rather than the Key Manager. For example, we've redacted last_name on the fly in our table:

last_name	gender	race	ssn
REDACTED	Female	Samoan	prMjCMmSo5g=:4vivaj7tmpGk+hjC0W0gaQ==
REDACTED	Female	Ute	VFK6lvJ1d68=:dz8wtuV0njXpFZIKmo95jg==
REDACTED	Male	Potawatomi	Mjc5GAAzULU=:4wkjKQI3WNrUEVWZexXtCA==
REDACTED	Female	Melanesian	TzzEyS2P2LA=:Jkid3z98cgcTik/TsKY2Pw==
REDACTED	Female	Micronesian	V3M7g+5GJJM=:OvRR8maUE4Dj+gW//o69XQ==
REDACTED	NULL	NULL	5EqSayRGKWs=:LNzY7gsCEr67i+3+e6Eh6Q==
REDACTED	Male	NULL	5zM5YISGHZc=:I/moeZ7fXWdo4cmr9HJdew==
REDACTED	Female	Argentinian	zAMBvSAqERo=:1TtOMmDPBJvTw3cBBDqv7A==
REDACTED	NULL	NULL	cleHnbAng44=:TUQsyC1Hh+E7ssWyY9LZUQ==
REDACTED	Female	Cherokee	8DM8JxM9r7U=:0ezhZPGqcWUkIM2GVX6sGw==

You are now at the intersection of security and privacy and meeting the demand of both your data analysts and your legal and compliance teams. But how do you actually implement it? This is where Immuta comes in. Immuta can act as that interface we've been discussing to execute dynamic masking and decryption. Below is a diagram of how it works:



As you can see, the typical analytical user on the right is querying the data as usual from their preferred cloud compute engine, and Immuta is dynamically masking the data (privacy controls) based on policy, natively in the engine. The encrypted SSN can never be decrypted natively in the cloud engine.

Should a user with special access need to see the SSN column decrypted, they can leave the native engine and query through the Immuta proxy, which lives on-premises (if desired). In this case, that interface is able to decrypt the data using a customer-provided encryption/decryption algorithm service. This can talk to the key management service – the details of which are completely abstracted from Immuta. Note that this on-premises proxy can also achieve the dynamic masking and can support any other database on-premises or in the cloud.

IMMUTA

115 Broad Street, 6th Floor, Boston, MA 021100 | immuta.com | (800) 655-0982