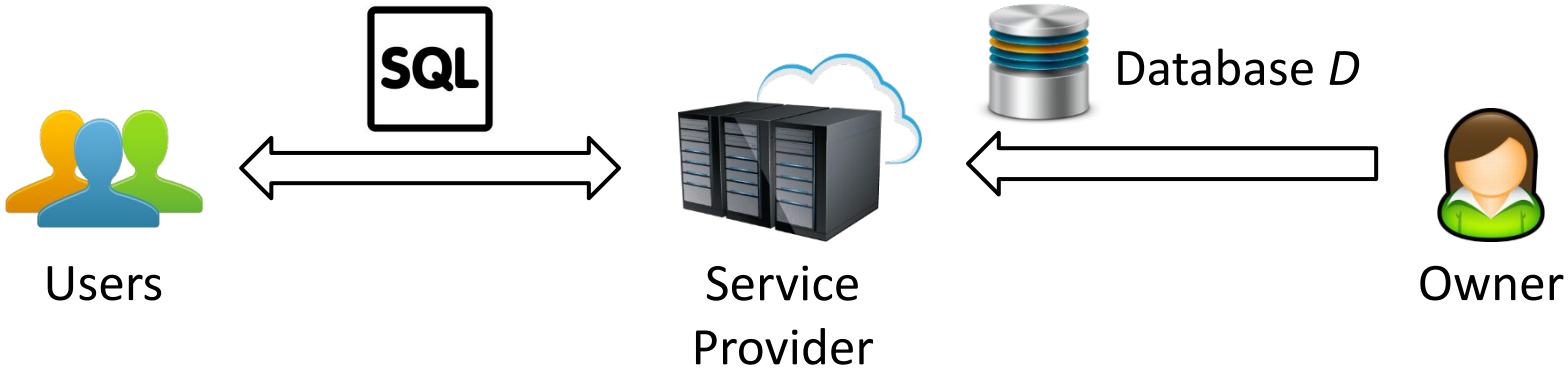
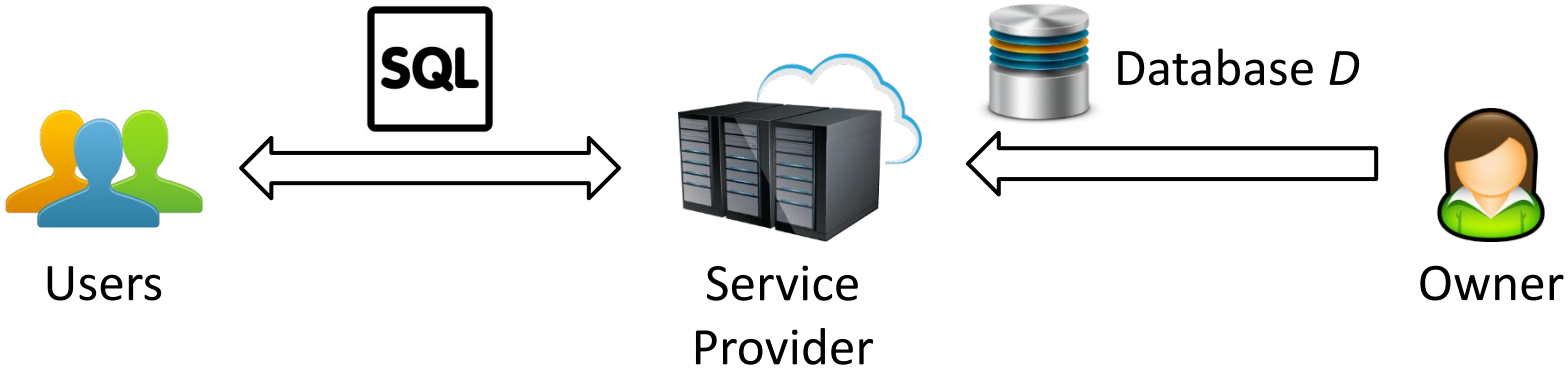
CS5322 Database Security

Query Authentication: Motivation



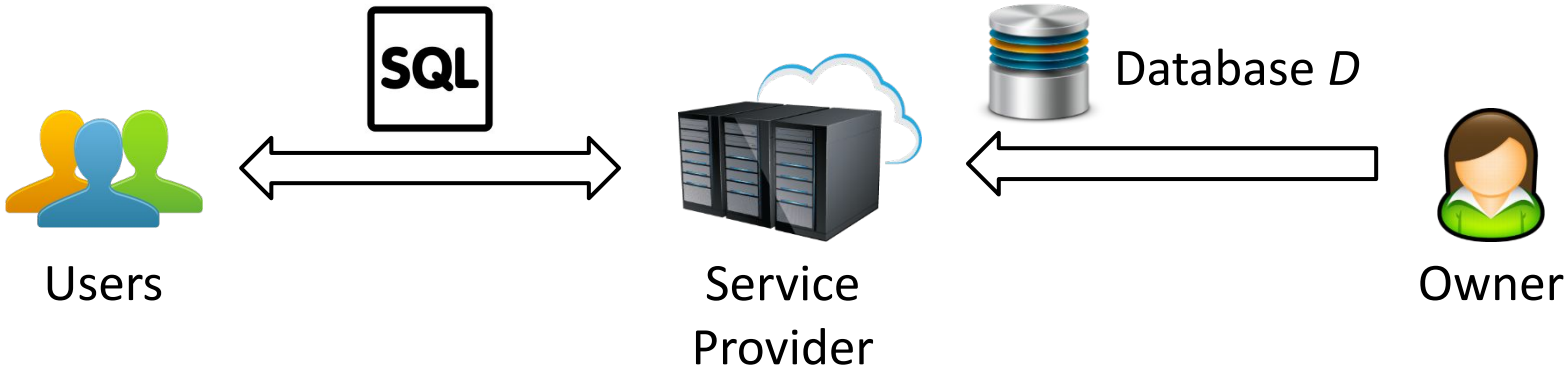
- A data owner passes a database *D* to a service provider
- The service provider answers queries from users
- This setting is referred to as *database outsourcing*

Query Authentication: Motivation



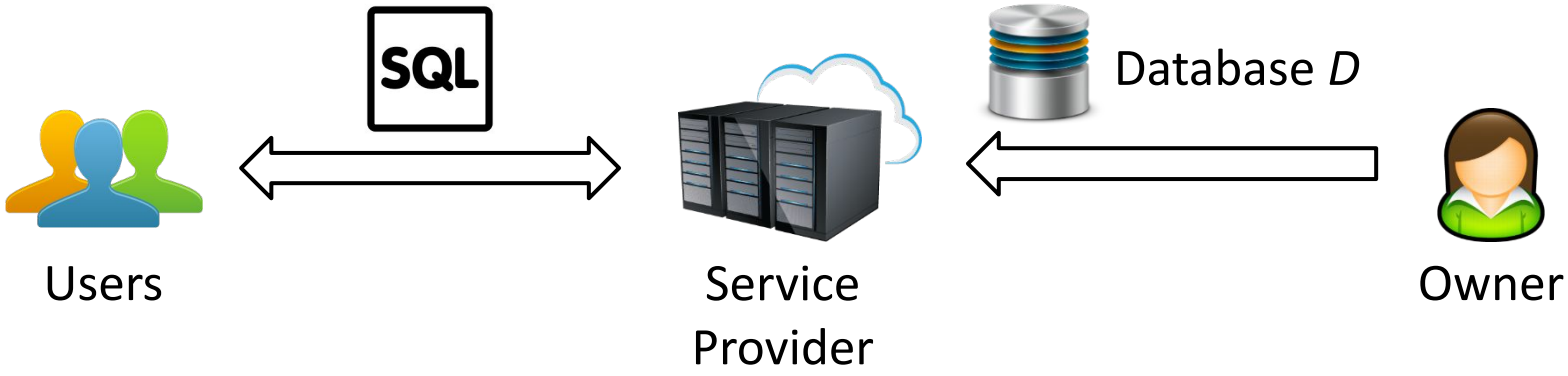
- This was the same scenario that we considered in our discussions of *encrypted databases*
- But now, let's assume that we do NOT aim to ensure data confidentiality against the service provider
  - Reason: encrypted databases incurs a lot of overheads
- Instead, we want to ensure that the service provider answers queries *correctly*

Query Authentication: Motivation



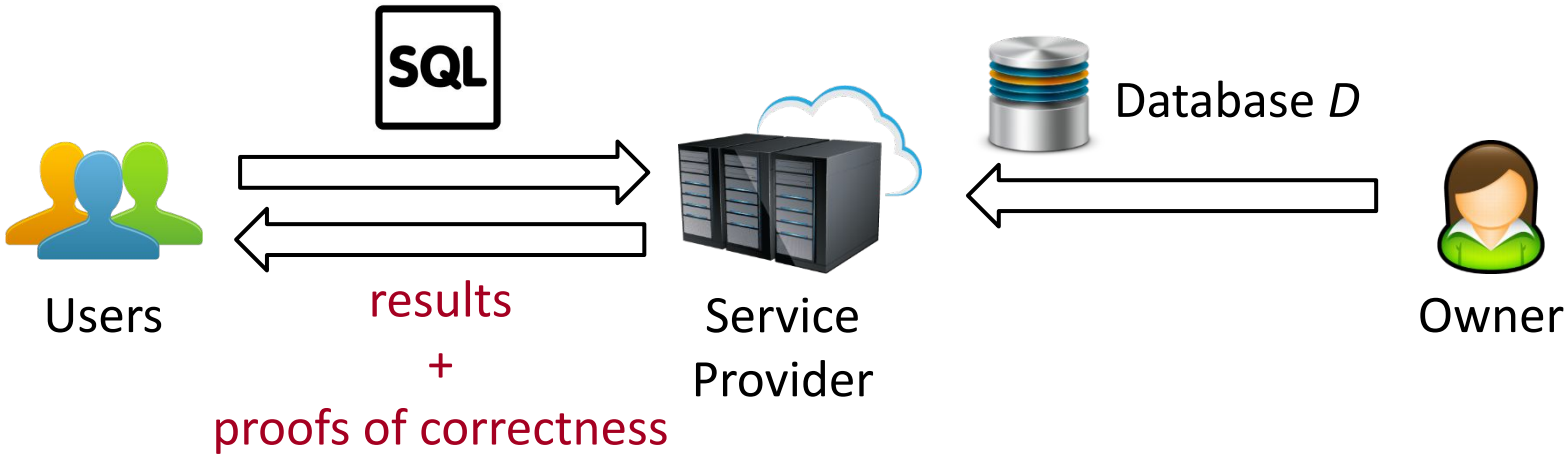
- "What do you mean by 'answering queries correctly'?"
- Possible ways to answer queries incorrectly:
  - Return only a subset of the tuples in the query result (e.g., by stopping the query early)
  - Return some fake tuples
  - Return the query result previously obtained from an outdated version of the database

Query Authentication: Motivation



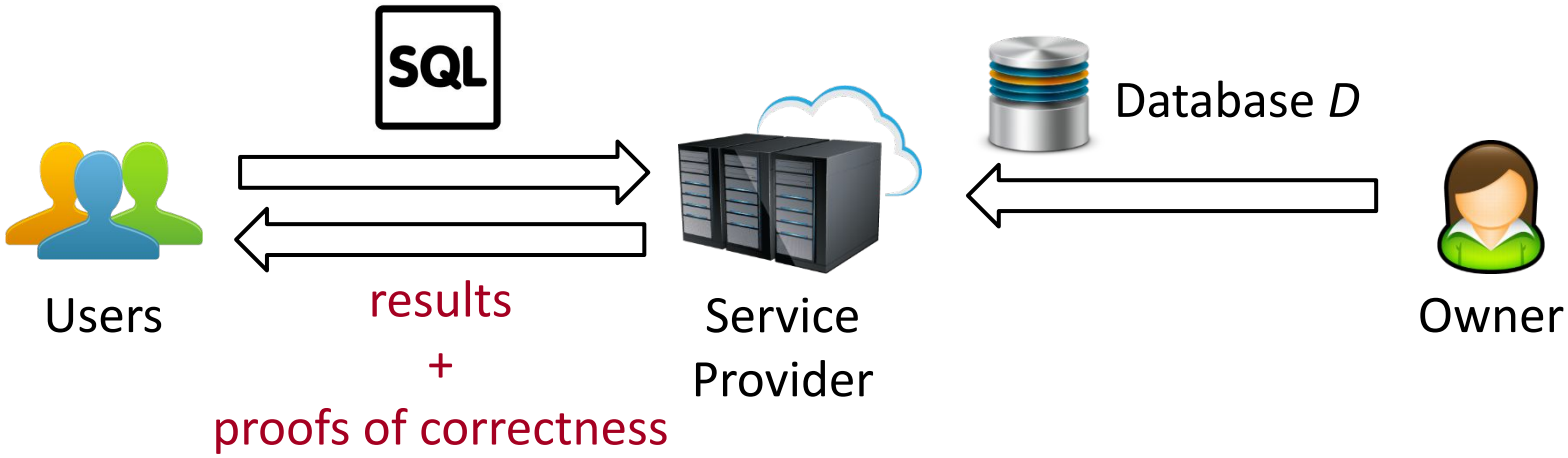
- "Why would the service provider do that? "
- Possible reasons:
  - The service provider is overloaded with a lot of queries on a lot of databases, and does not have enough resource to process all queries in time
  - The service provider wants to compromise the query result for malicious purposes
  - …

Query Authentication: Motivation



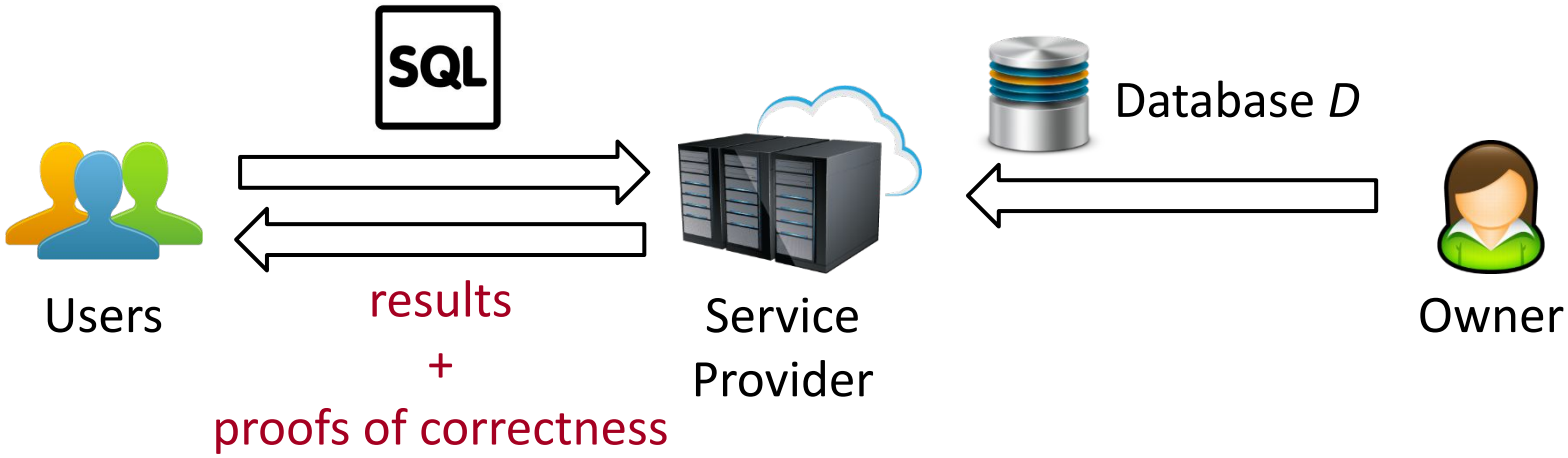
- Solution:
  - When the service provider answers a query, it returns both the query result and a proof of the result's integrity
  - i.e., the proof should let the user verify whether the result is correct and whether it is from the most updated database

Query Authentication: Motivation



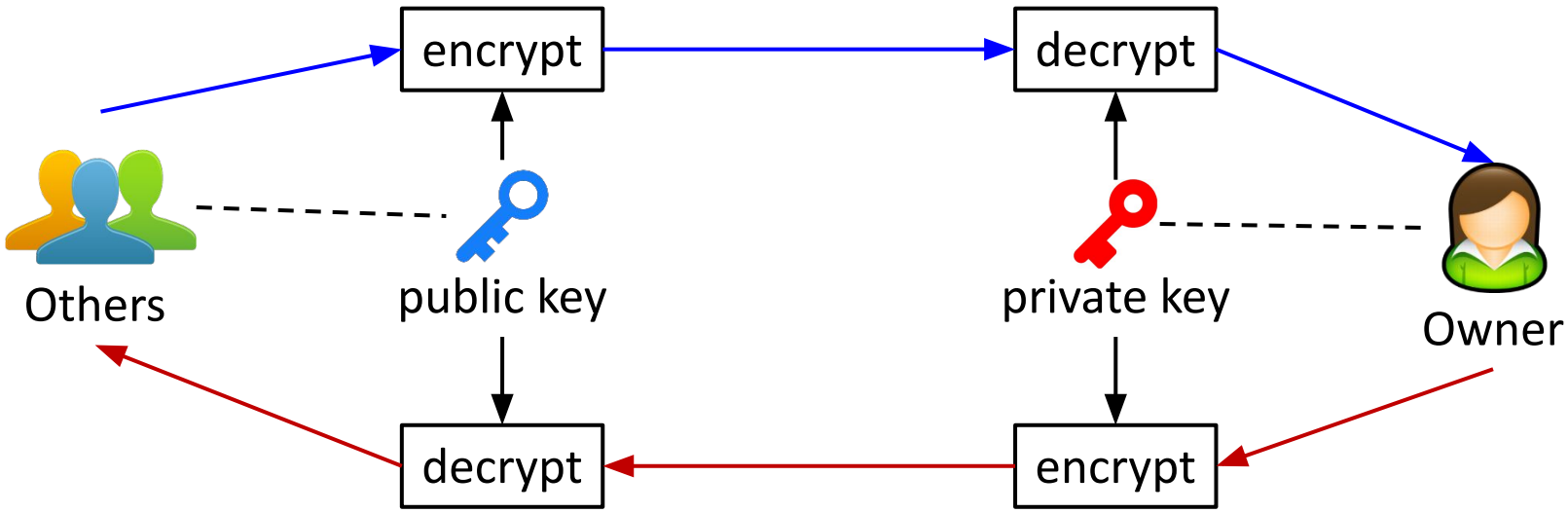
- "I don't think service providers would really do that in practice."
- Amazon has done something based on a similar idea
  - https://aws.amazon.com/qldb/
- Microsoft also proposed something along a similar line
  - https://dl.acm.org/doi/10.1145/3035918.3064030

Query Authentication: Motivation



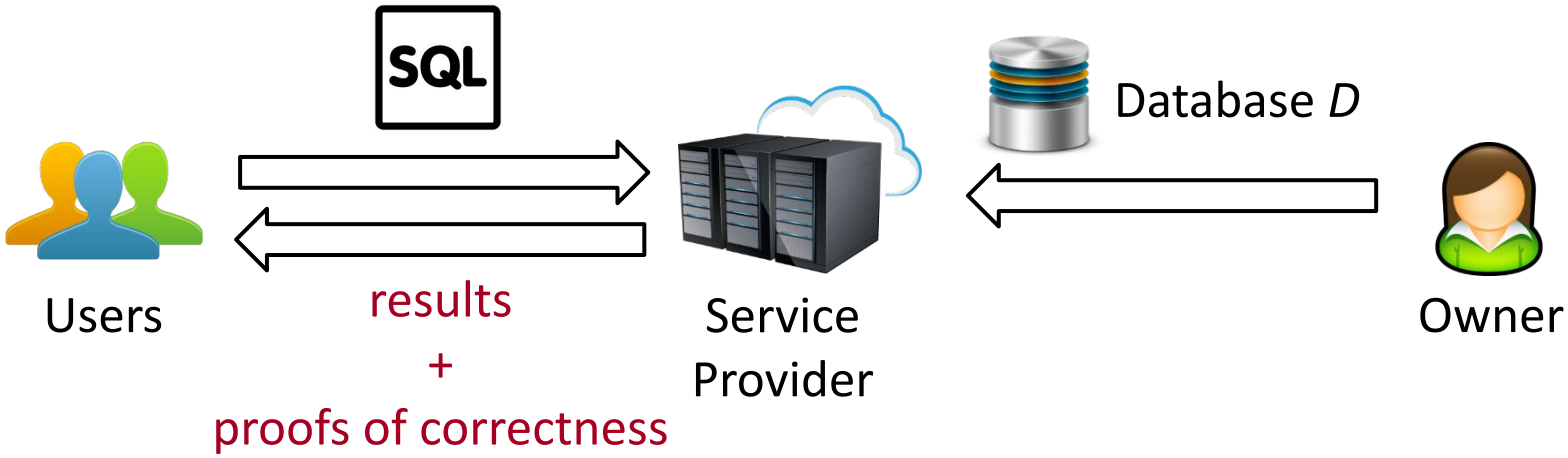
- "OK. But how?"
- Use *public-key cryptography*

Public-Key Cryptography



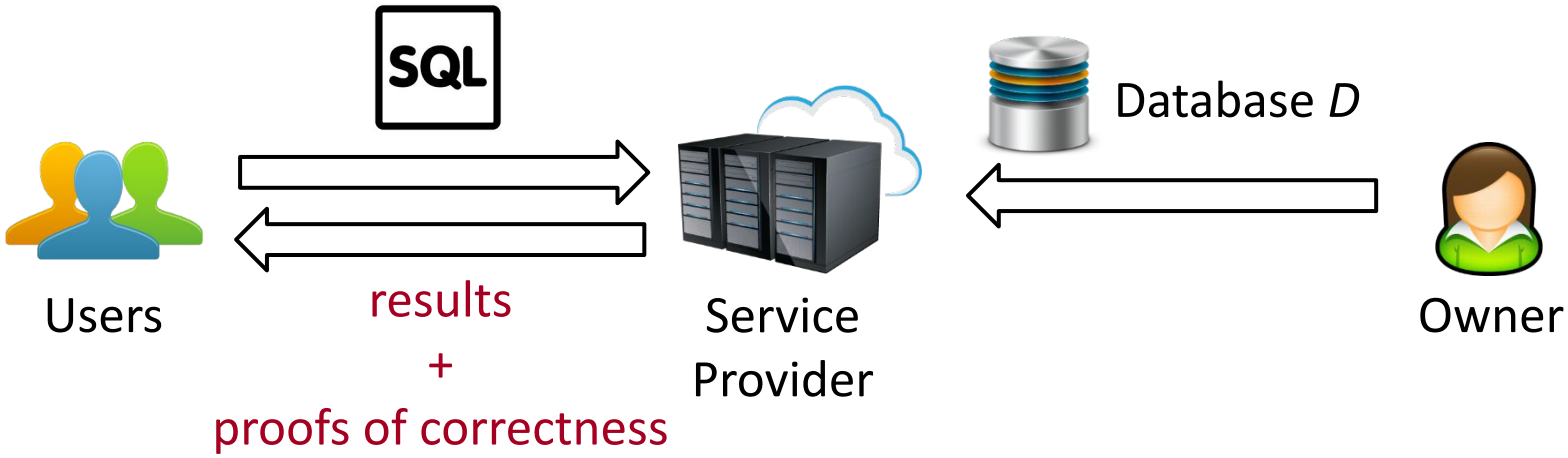
- The data owner has two keys
  - A private key *sk*, which she keeps to herself
  - A public key *pk*, which she distributes to the public
- Messages encrypted by *pk* can only be decrypted by *sk*
  - So anyone can use *pk* to send secret messages to the owner
- Messages that can be decrypted by *pk* must have been encrypted by *sk*
  - So the owner can send *authenticated* messages to others

Naïve Solution



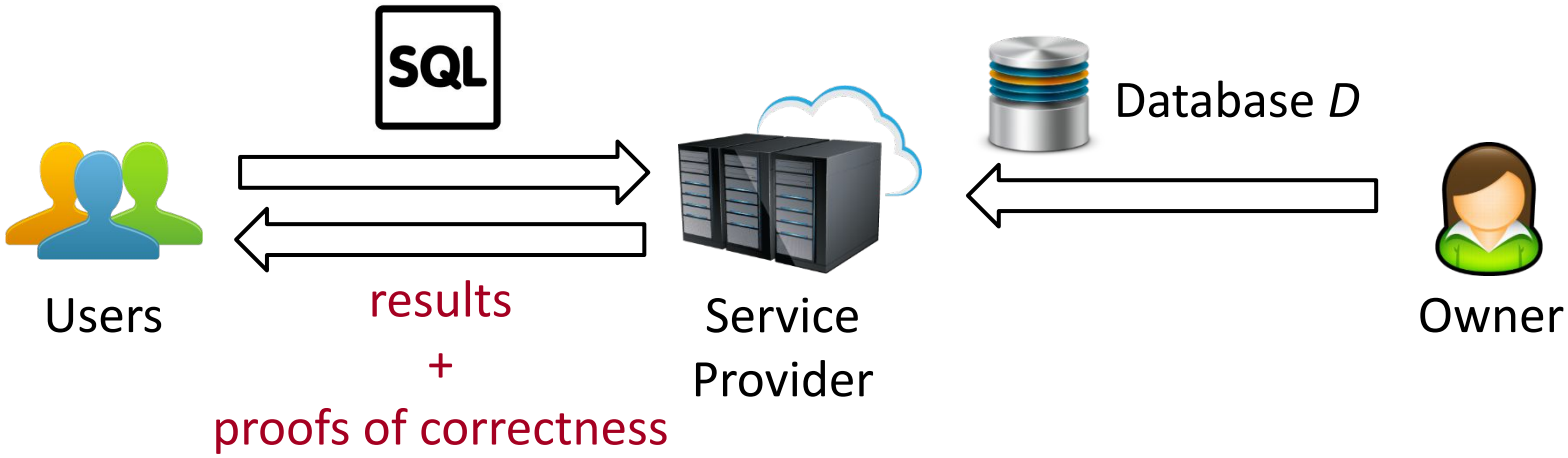
- Let the owner *sign* each tuple in the database, i.e.,
  - She encrypts each tuple with her private key *sk*
  - She sends both the encrypted version and the original to the service provider
- Whenever the service provider answers a query by returning some tuples, he also returns the encrypted versions

Naïve Solution



- Whenever the service provider answers a query by returning some tuples, he also returns the encrypted versions
- For each tuple *t* and its encrypted version *t**, the user can decrypt *t** using the owner's public key, and see if the decrypted tuple is the same as *t*
  - This prevents the service provider from modifying or faking any tuple

Naïve Solution

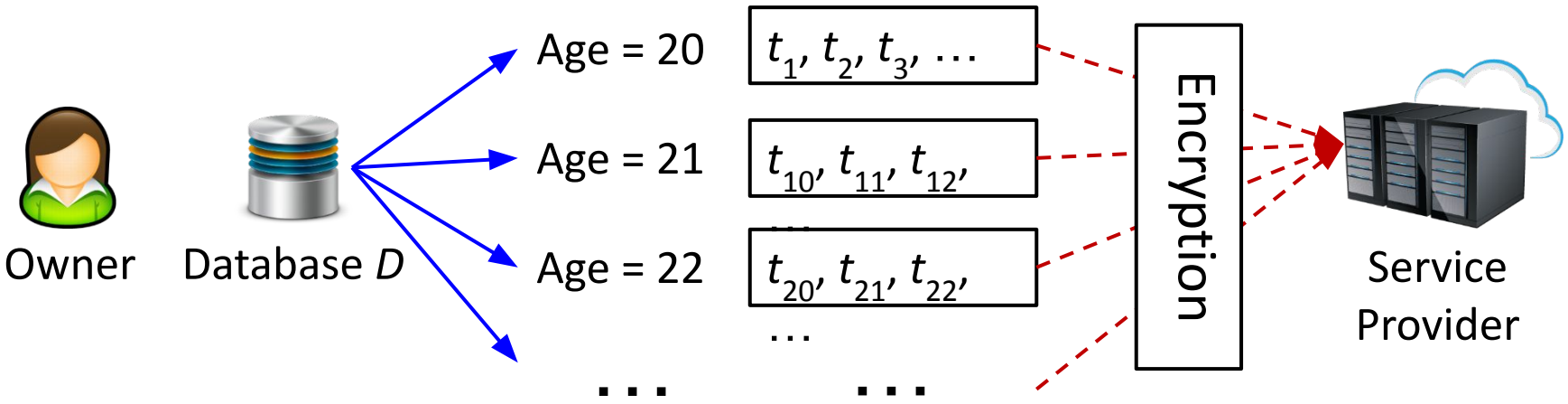


- Problems:
  - It does not prevent the service provider from dropping any tuples
  - Each tuple needs to be stored twice: encrypted and unencrypted
- We need something better…

Towards An Improved Solution

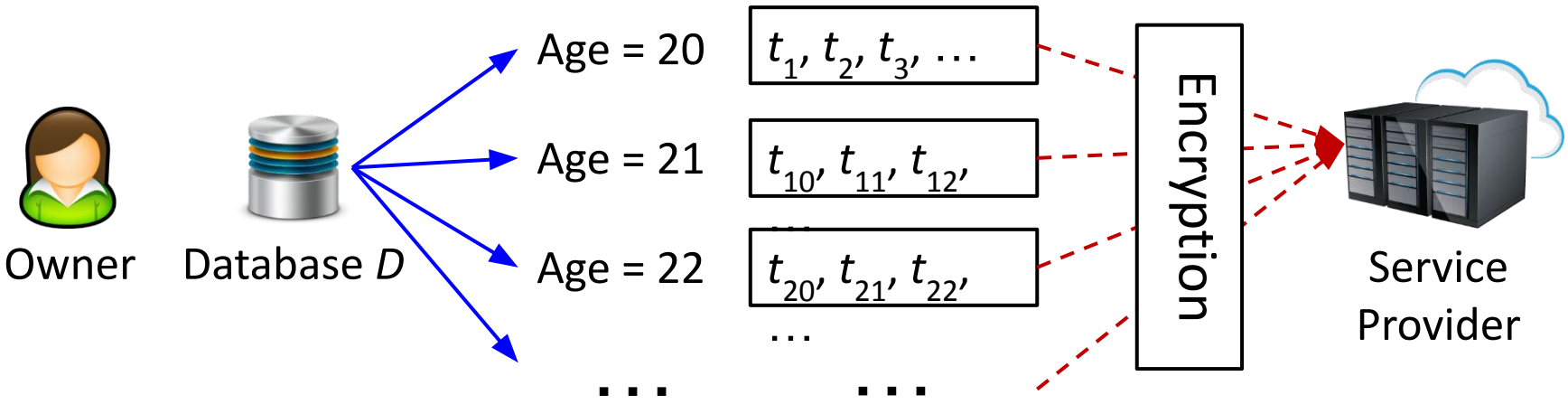
- Let's focus on a specific type of queries: equality query on one attribute
  - i.e., SELECT * FROM *T*
    WHERE *T.A* = *X*
  - e.g., SELECT * FROM Employees
    WHERE Age = 30
- How can we outsource this type of queries and prevent the service provider from faking or dropping results?

Towards An Improved Solution



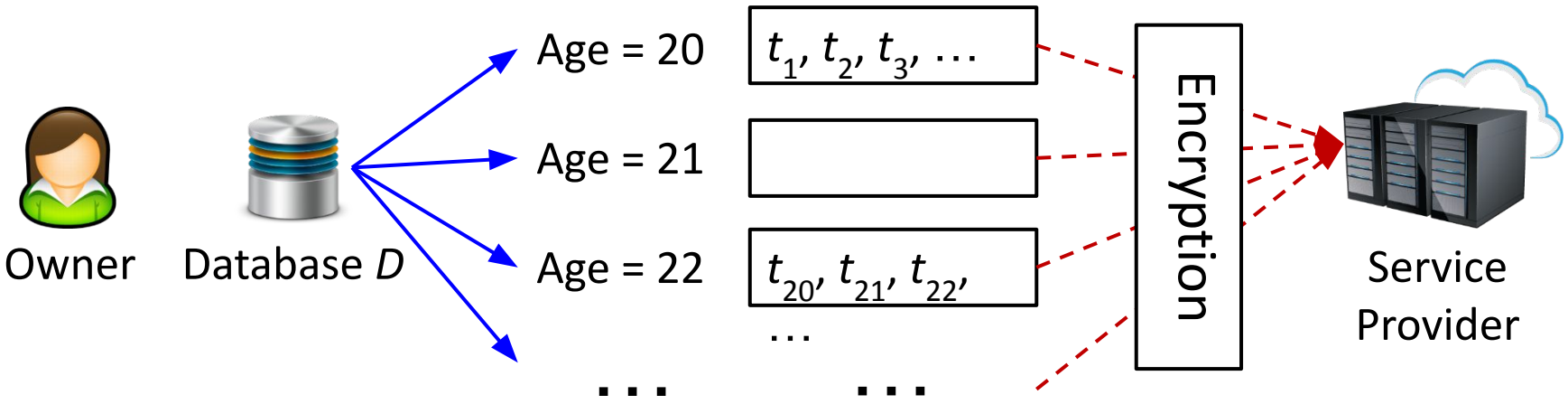
- Idea: We let the data owner
  - Divide the tuples into groups according to their Age values
  - Encrypt each group separately using her private key *sk*
  - Send the encrypted groups to the service provider

Towards An Improved Solution



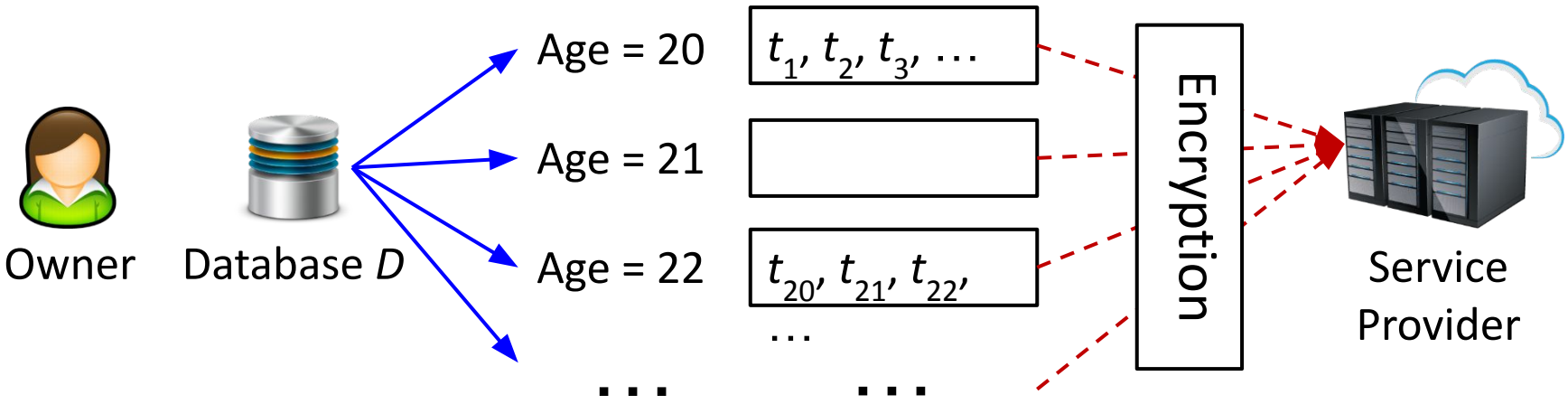
- Suppose that the user issues the following query:
  - SELECT * FROM Employee WHERE Age = 21
- The service provider just returns the encrypted group of tuples for Age = 21
- The user then decrypts the tuple group using the data owner's public key *pk*
  - If the tuples have Age = 21, then it is guaranteed that there is no fake or missing tuples

Towards An Improved Solution



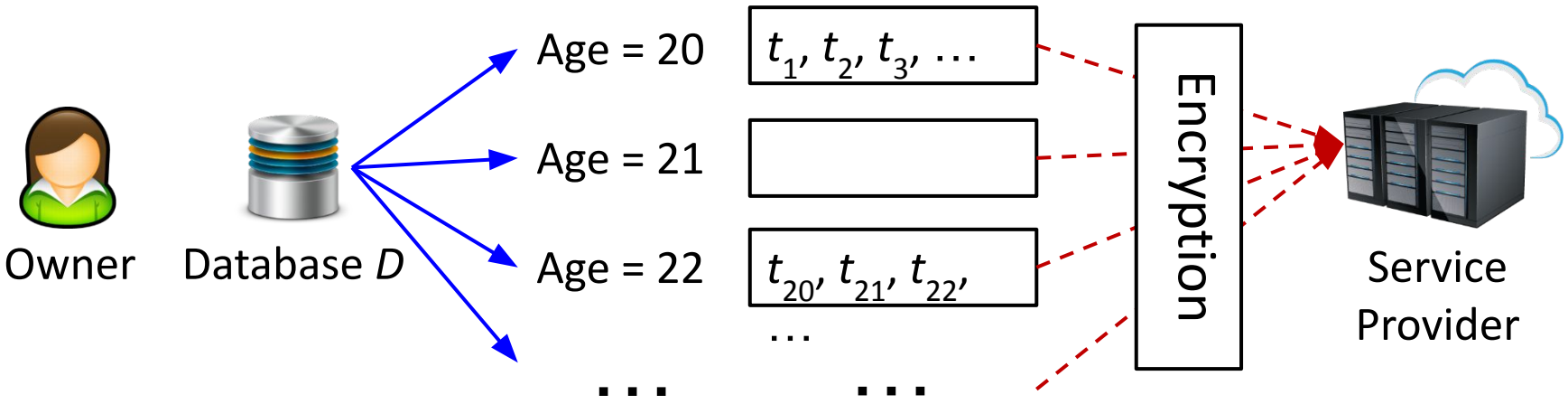
- Question:
  - Suppose that there is no employee with Age = 21
  - What does the data owner do?
- Option 1:
  - Just omit the Age = 21 group
  - Only create encrypted groups for those age values that have at least one tuple
- Does this work?

Towards An Improved Solution



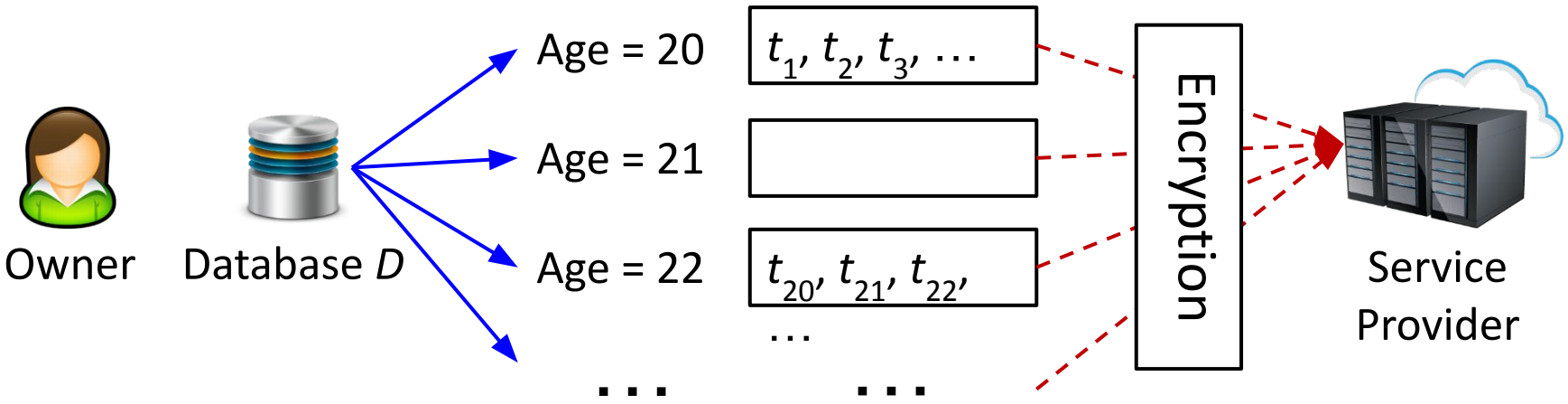
- Option 1:
  - Only create encrypted groups for those age values that have at least one tuple
- Does this work?
- No
  - If we do this, then the service provider can drop any tuple group from the query result, by pretending that there is no tuple with that age value

Towards An Improved Solution



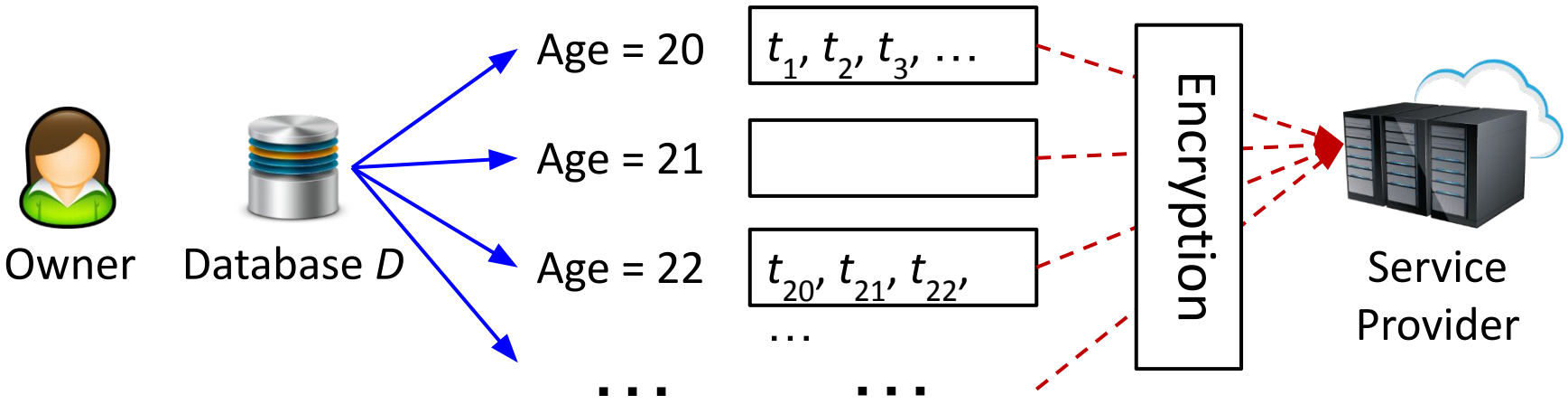
- Option 2:
  - If Age = 21 does not have any tuple, then the data owner encrypts a message "Age = 21 is empty" using her private key
  - Do this for every empty Age group
- If a user asks for an Age group that is empty, just return the corresponding encrypted message

Towards An Improved Solution



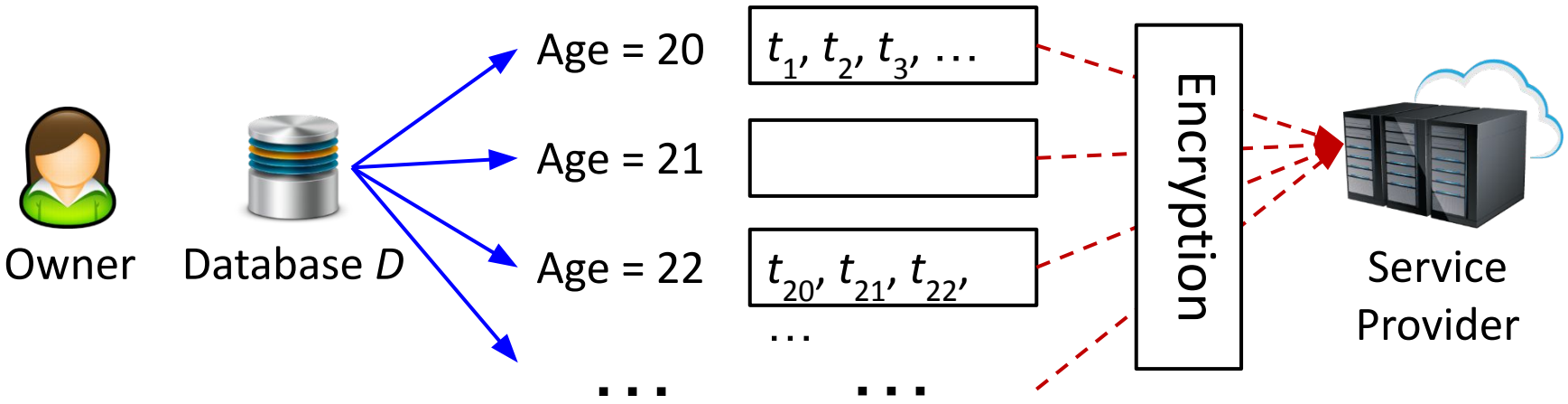
- Option 2: Create an encrypted message for each empty group
- Problem: There could be too many empty groups
  - Suppose that the attribute to be queried is Salary
  - Creating an encrypted group or message for each possible Salary value is rather expensive in terms of time and space

Towards An Improved Solution



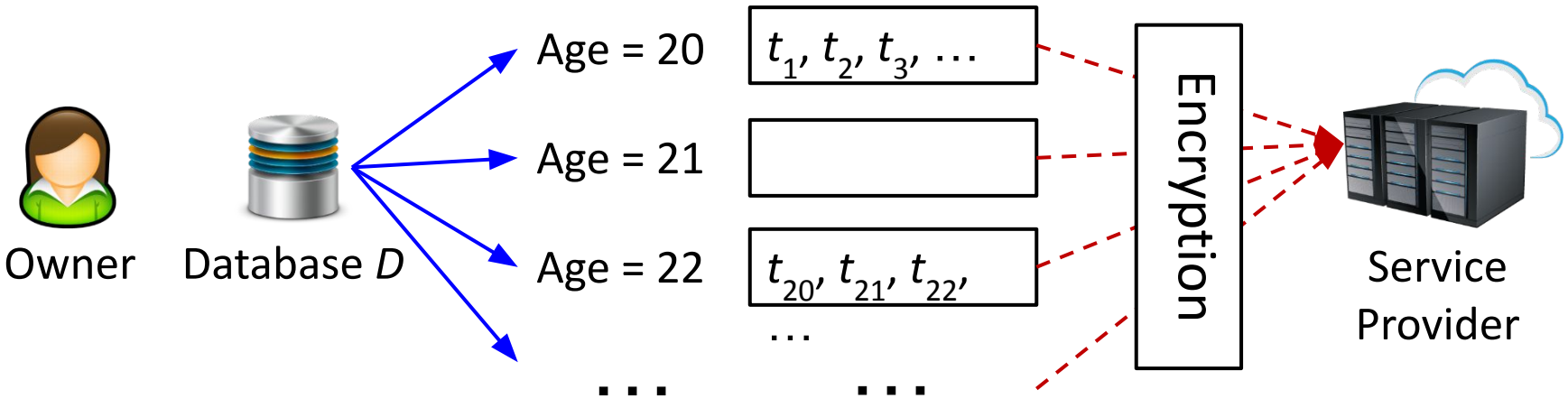
- Option 3:
  - For each non-empty group, we not only encrypt the tuples, but also create an encrypted message to indicate which is the next non-empty group
  - E.g., in the above example, we not only encrypt the tuple group with Age = 20, but also create an encrypted message: “After Age = 20, the next non-empty group is Age = 22”
  - The total number of encrypt messages equals the total number of non-empty groups

Towards An Improved Solution



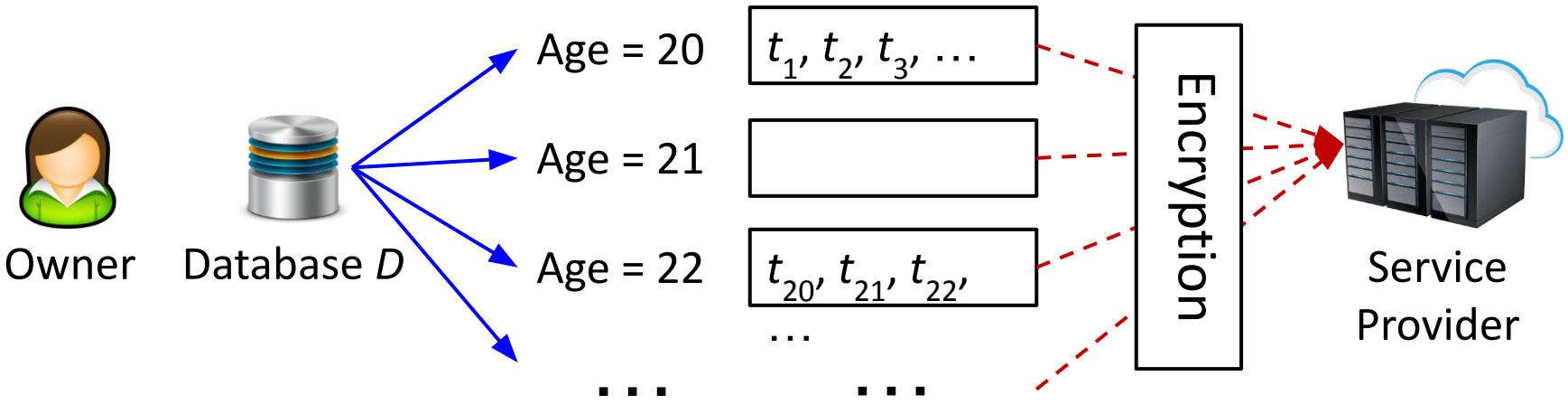
- If the user queries a non-empty group, then returns the corresponding encrypted group
- If the user queries an empty group, return the encrypted message that can prove the empty results
  - E.g., if the user enquires “Age = 21”, then return the encrypted message “After Age = 20, the next non-empty group is Age = 22”

Towards An Improved Solution



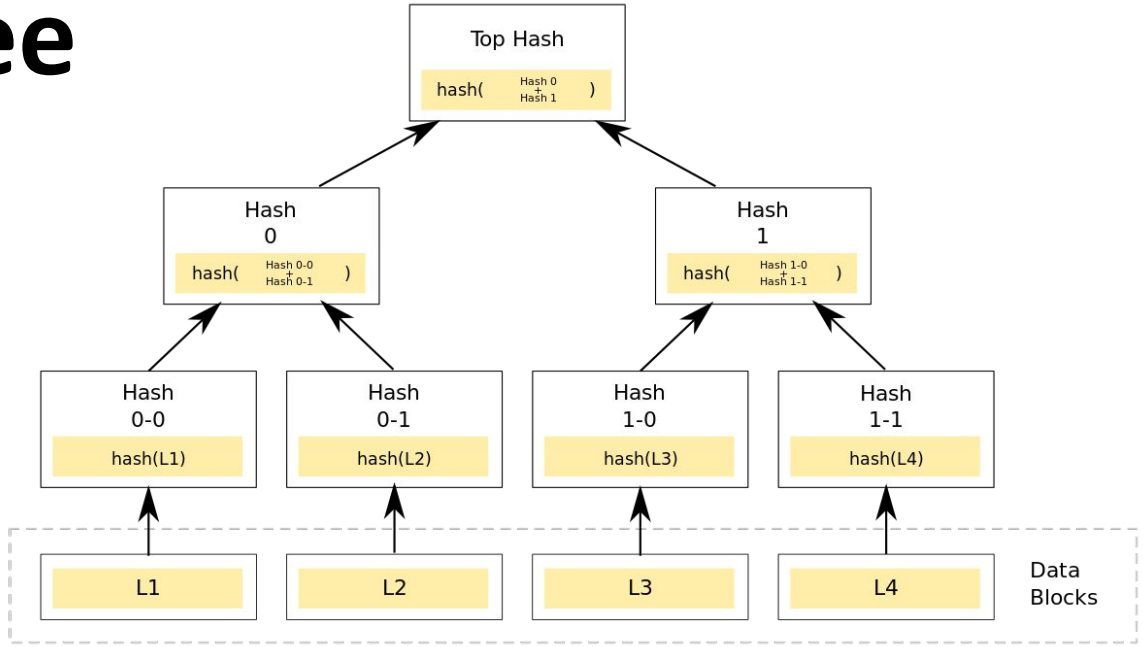
- Any problem?
  - There could be too many non-empty groups to encrypt
    - Think about Salary
  - Only works well for equality query
    - Does not work for range queries, e.g., “Salary > 100k and Salary < 200k”

Towards An Improved Solution



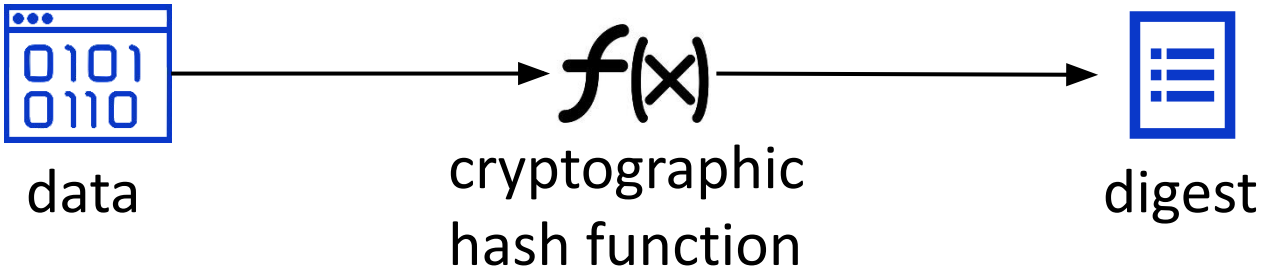
- We will introduce a solution that only require the data owner to create one encrypted "message"
  - No need to encrypt any tuple at all
  - And it can support range queries
- It is called the *Merkle tree*

Merkle Tree



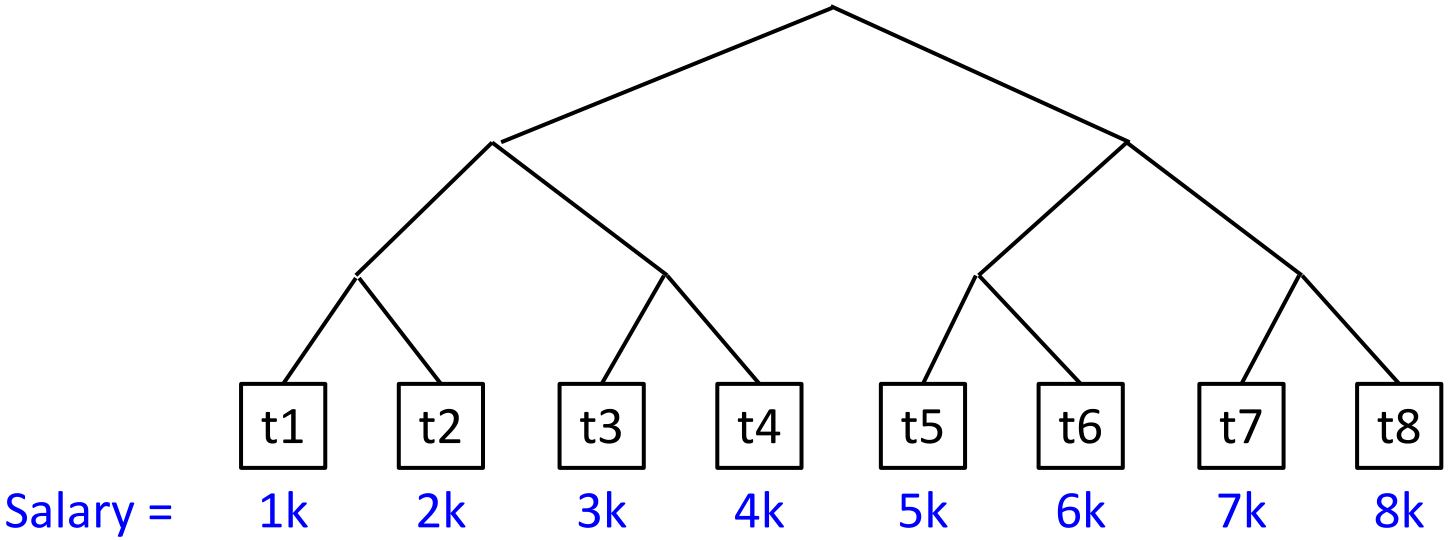
- Invented by Ralph Merkle in 1979
- A tree structure that allows efficient verification of its content
- Key ingredient: a cryptographic hash function

Cryptographic Hash Function



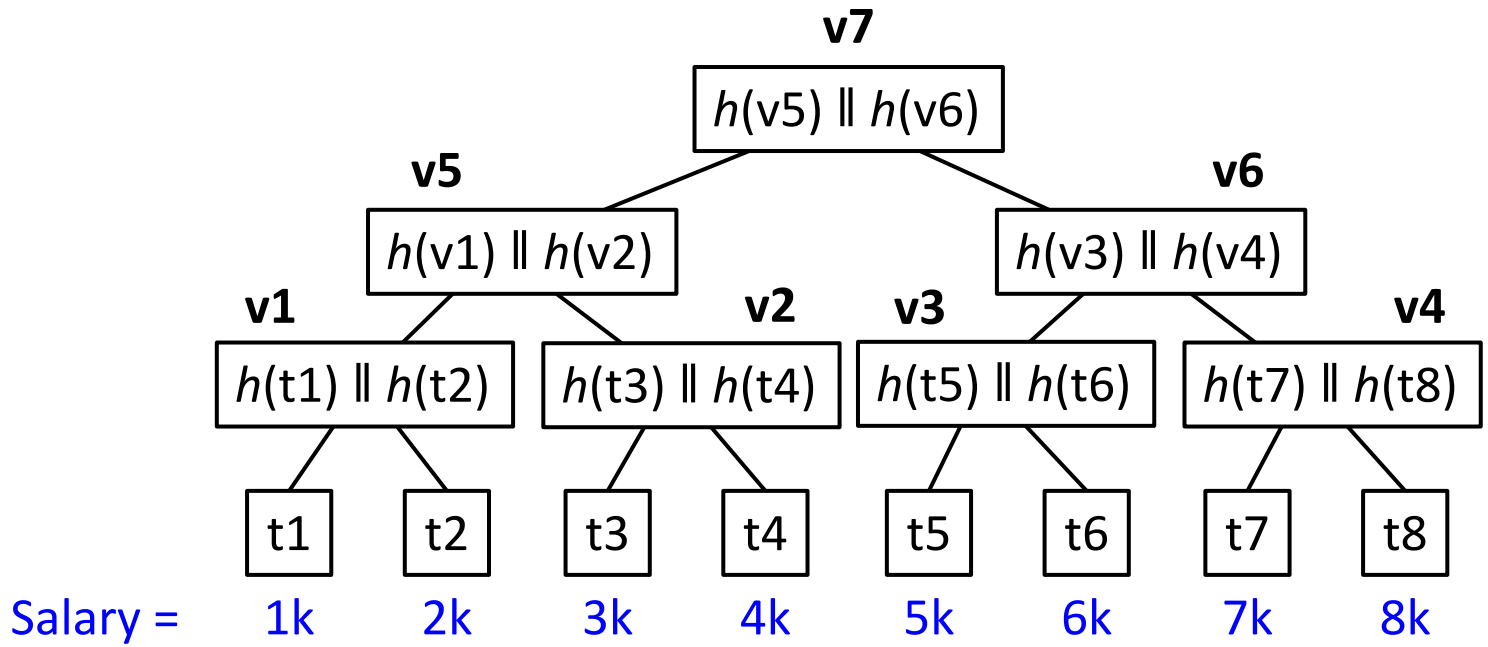
- A function that maps data of arbitrary size to a bit string of a fixed size, with the following properties:
  - It is deterministic (so the same message always results in the same hash)
  - It is efficient to compute the hash value for any given message
  - It is infeasible to generate a message from its hash value except by trying all possible messages
  - A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
  - It is infeasible to find two different messages with the same hash value

Merkle Tree: Example



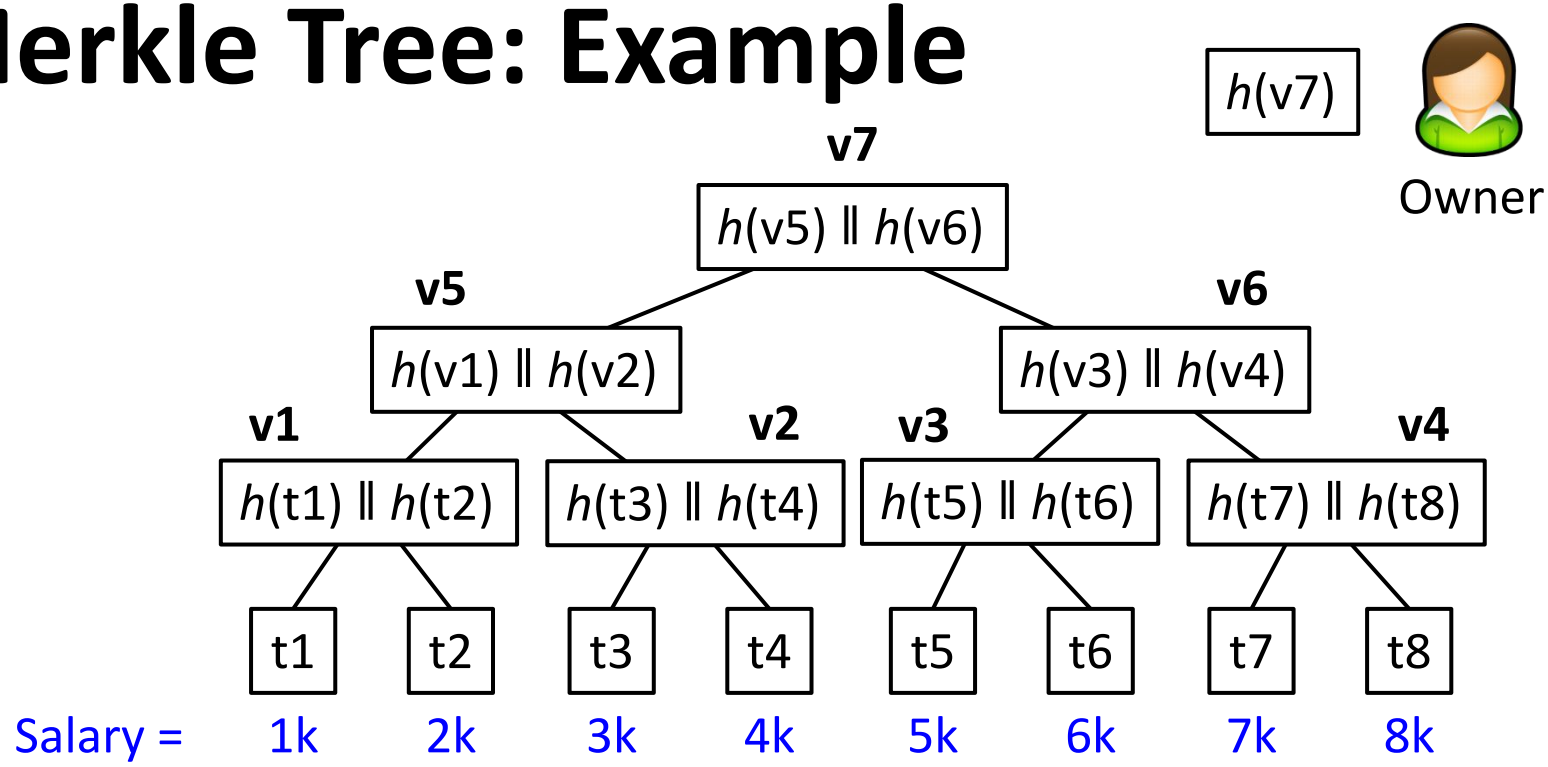
- Suppose that we are to build a Merkle tree to support range queries on Salary
- First, sort all tuples by Salary
- Second, build a binary tree on the sorted sequence

Merkle Tree: Example



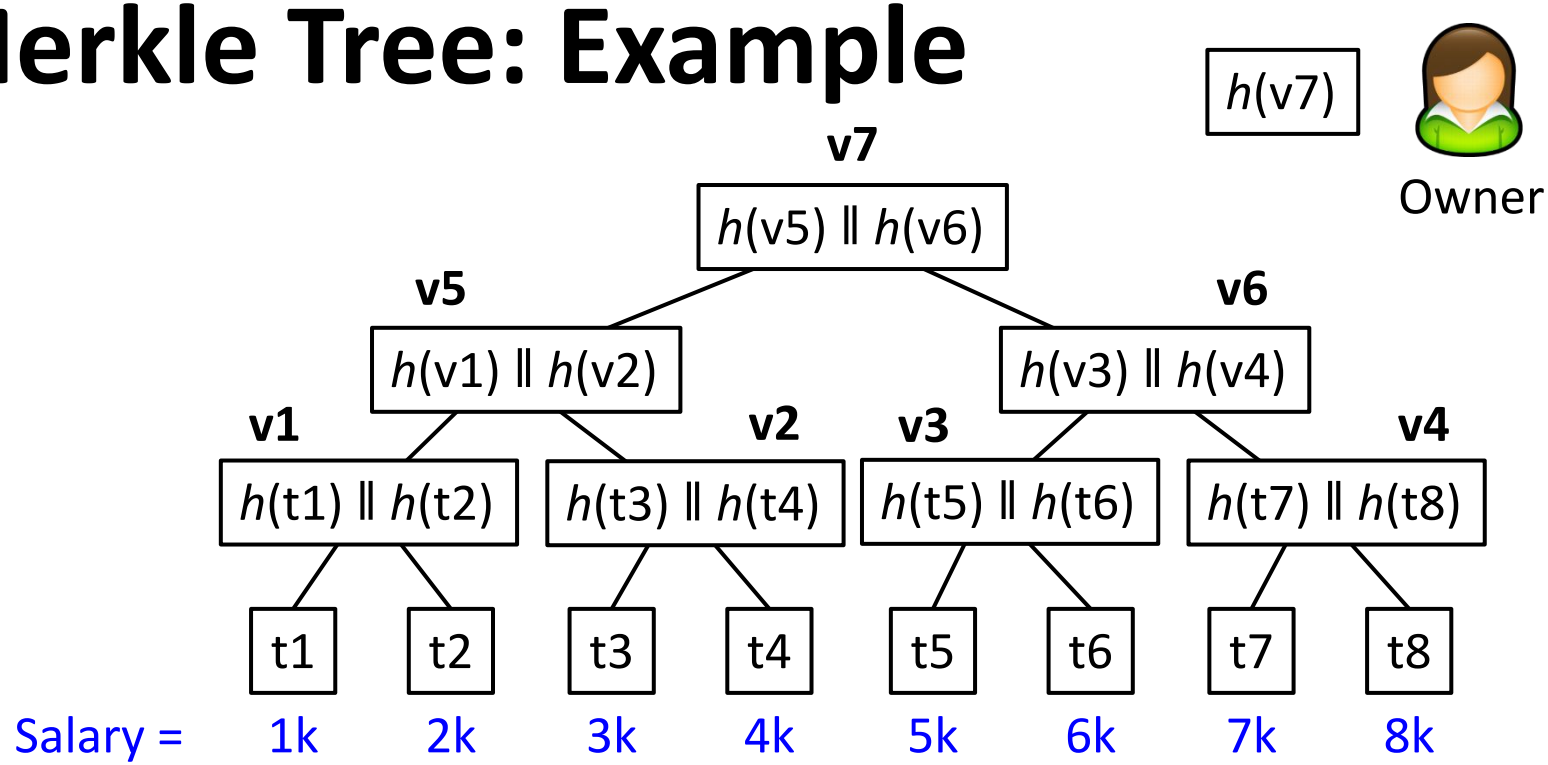
- Third, materialize the non-leaf nodes in a bottom up manner, using a cryptographic hash function *h*
  - For each non-leaf *v*, its content equals $h(v_{left}) \| h(v_{right})$, where v_{left} and v_{right} are *v*'s left and right children, respectively, and ‖ denotes concatenation

Merkle Tree: Example



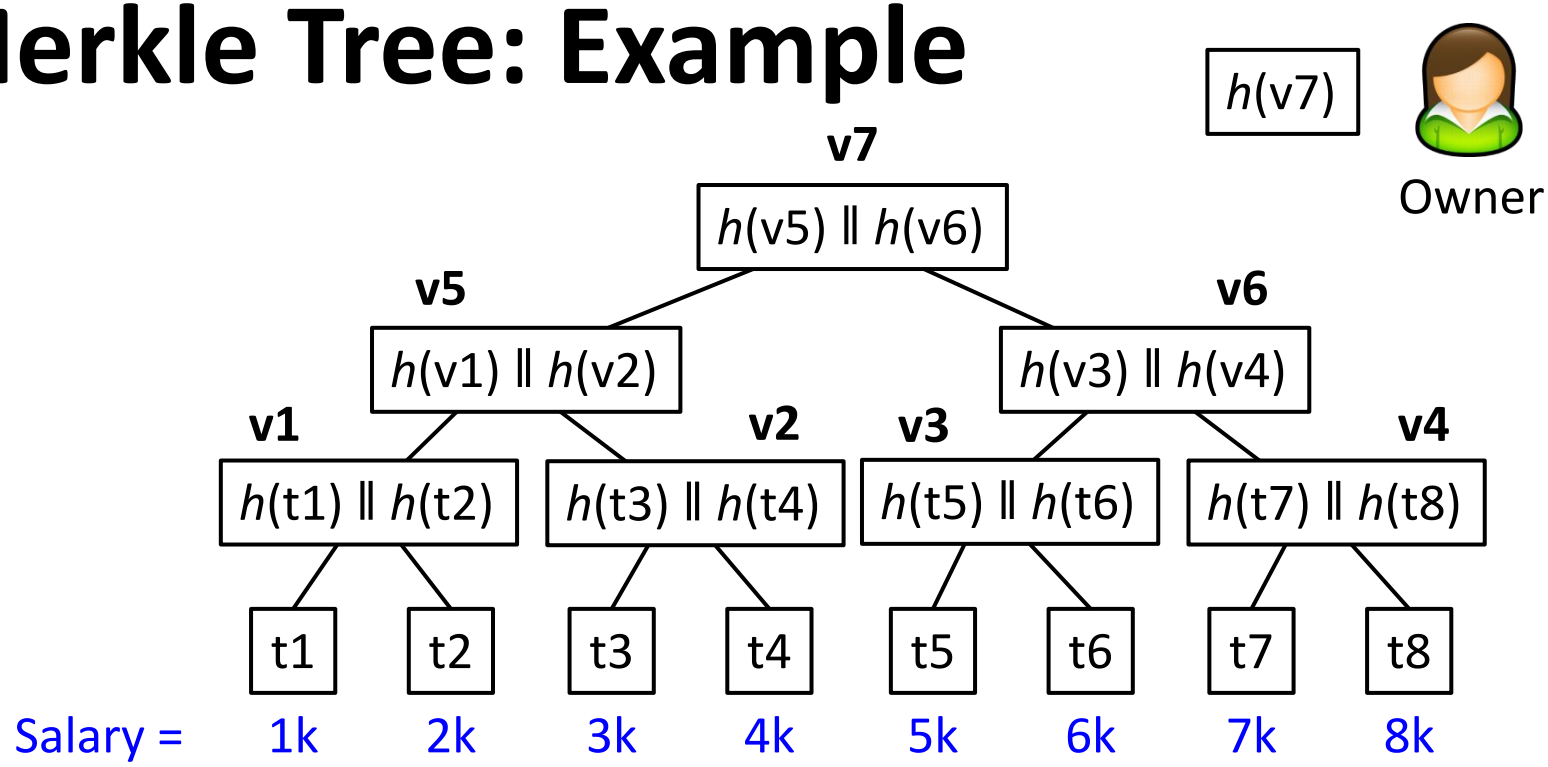
- Finally, let the data owner encrypts *h*(root) using her private key *sk*
  - In the above example, the root is v7
- Then, the data owner sends the encrypted digest to the service provider

Merkle Tree: Example



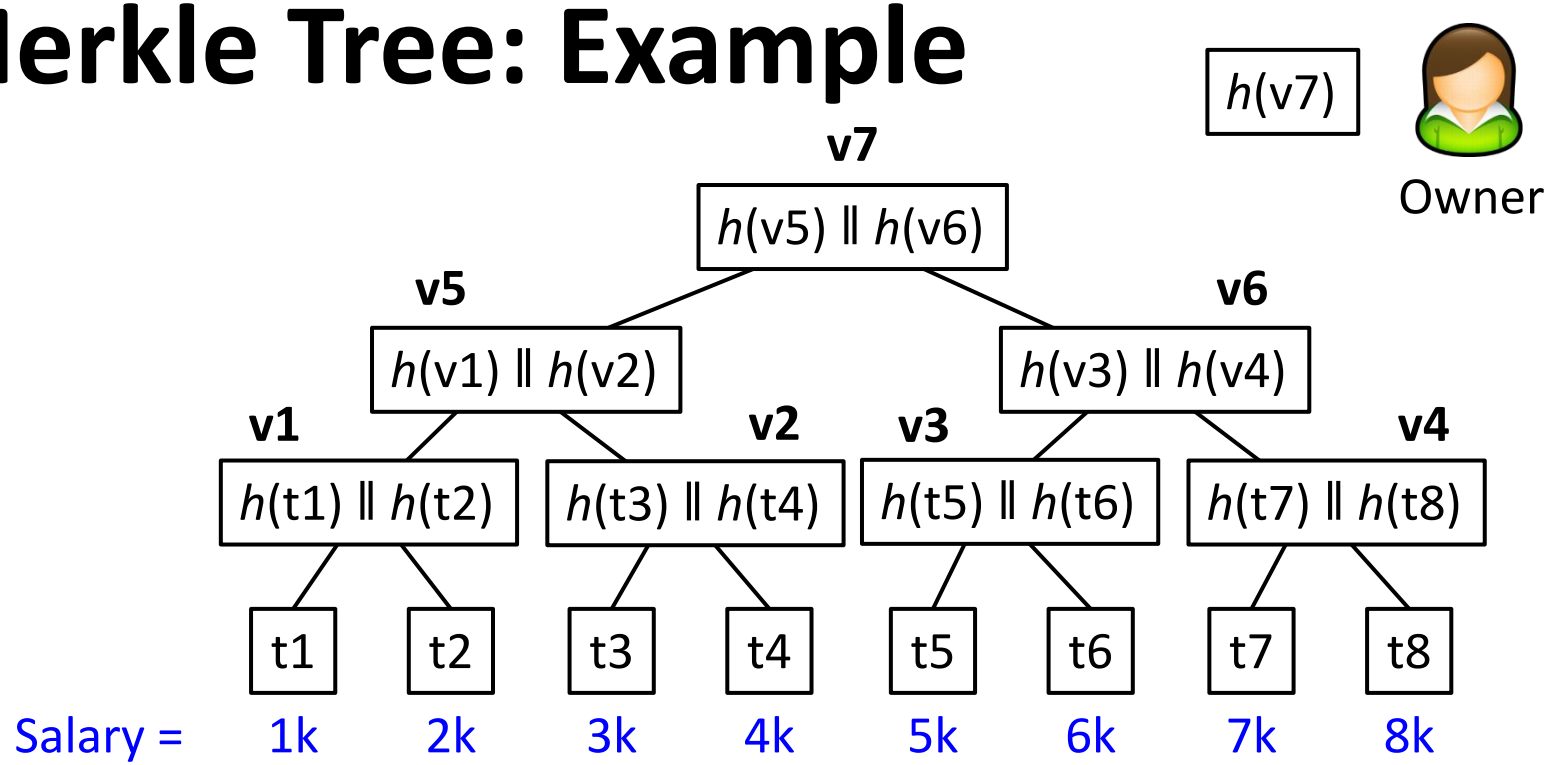
- Intuition:
  - The encrypted *h*(v7) ensures that the service provider cannot make any change to the sorted sequence t1, t2, …, t8

Merkle Tree: Example



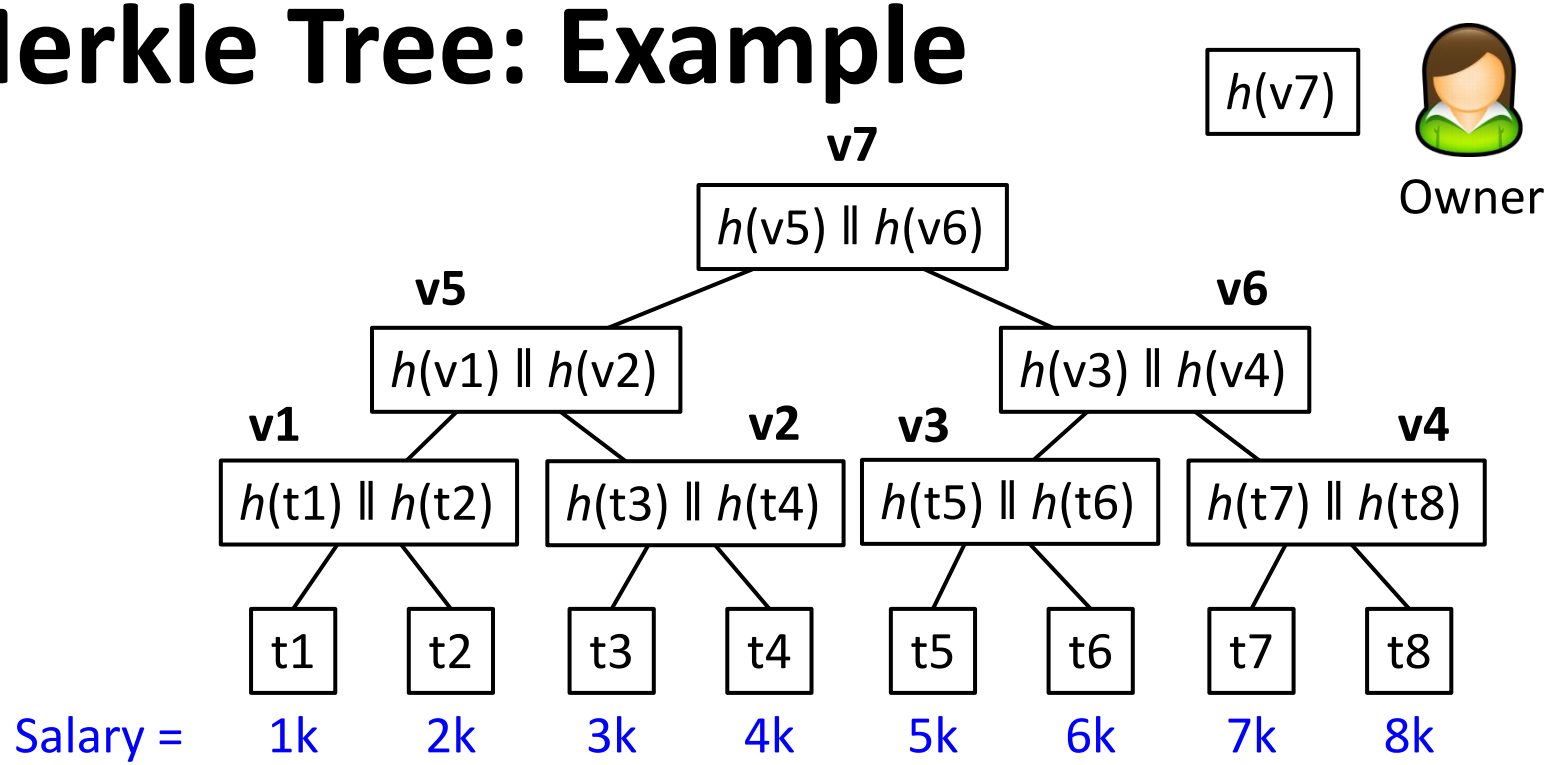
- Why?
  - Since *h*(v7) is signed by the data owner, the service provider will get caught if he changes v7
  - Since v7 cannot be changed, the service provider will get caught if he changes v5 or v6
  - Since v5 and v6 cannot be changed, the service provider will get caught if he changes v1, v2, v3, or v4, and so on…

Merkle Tree: Example



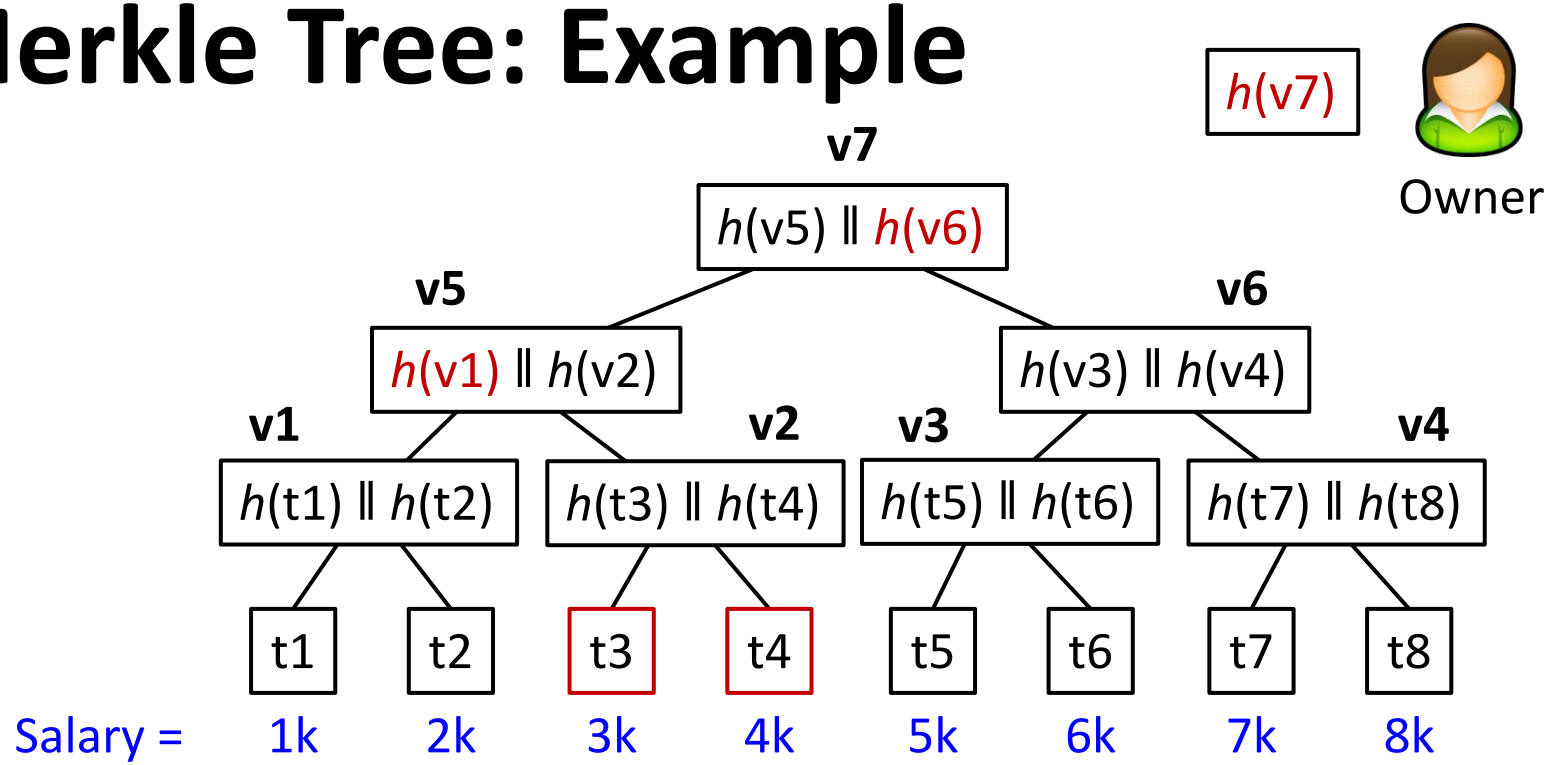
- In other words, the service provider can answer any query as follows
  - Return the sorted sequence t1, t2, t3, …, t8, along with the signed *h*(v7)
  - Ask the user to verify the correctness of the sorted sequence, and then answer the query herself using the sorted sequence
- Problem: this approach returns too many irrelevant tuples
- Solution: return some hash values instead of tuples

Merkle Tree: Example



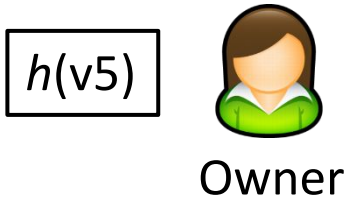
- Suppose that a user searches for "Salary = 3.5k"
- The service provider would return the following
  - t3, t4, h(v1), h(v6), and the encrypted h(v7)

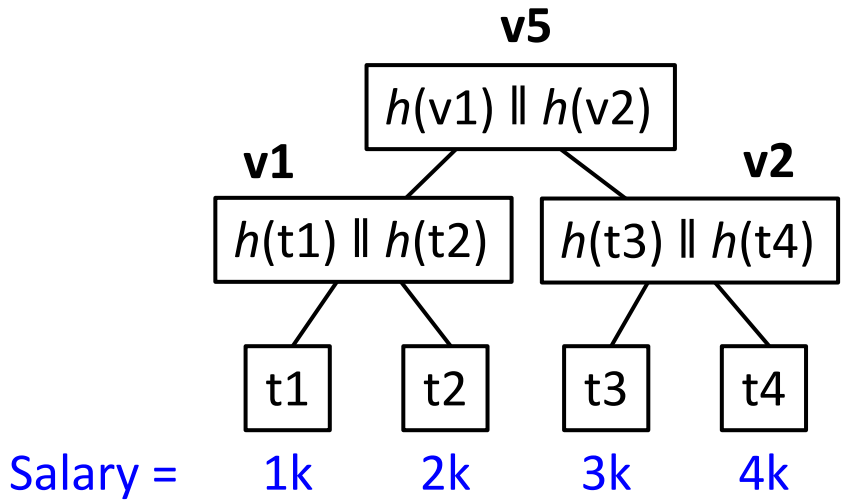
Merkle Tree: Example



- Suppose that a user searches for "Salary = 3.5k"
- The service provider would return the following
  - t3, t4, h(v1), h(v6), and the encrypted h(v7)
- Why? We will explain using a simpler tree

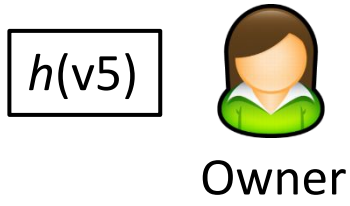
Merkle Tree: Example

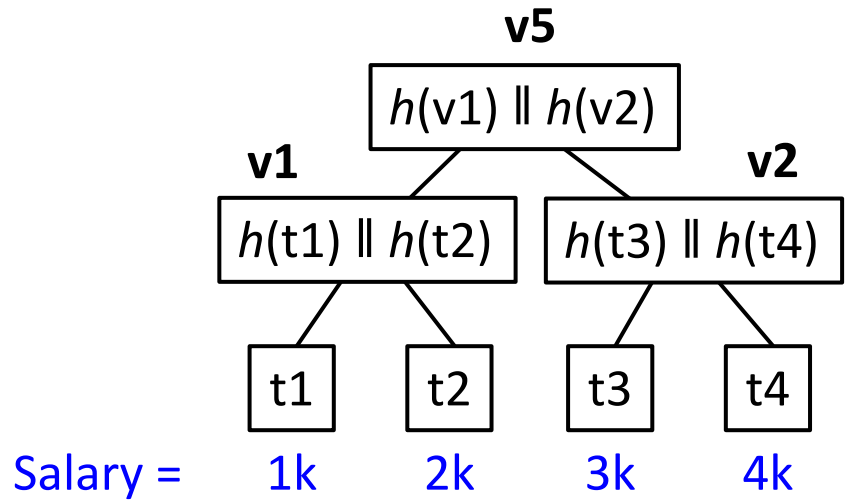




- Consider the above Merkle tree
- Re-consider the query on "Salary = 3.5k"
- Option 1:
  - The service provider could return t1, t2, t3, t4, as well as the encrypted *h*(v5)
  - The user could then compute *h*(t1), *h*(t2), *h*(t3), *h*(t4)
  - Based on that, she computes *h*(v1) and *h*(v2)
  - Then she can compute the hash of *h*(v1) ‖ *h*(v2) and verify it against the encrypted *h*(v5)
  - Then she can be sure that the data has only t1, t2, t3, and t4; so no "Salary = 3.5k"

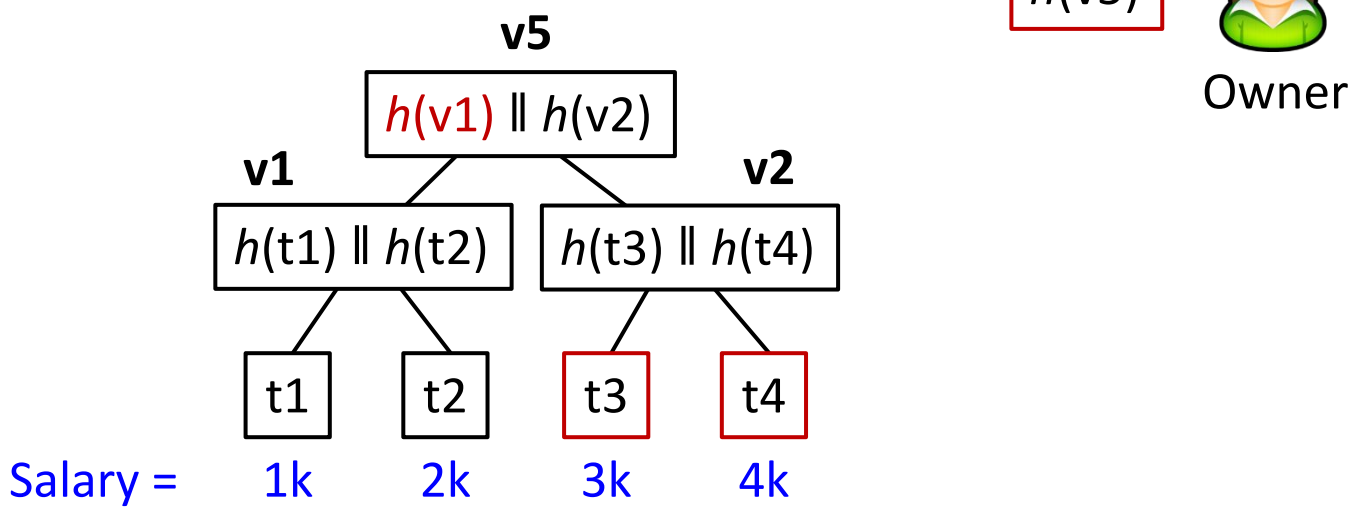
Merkle Tree: Example





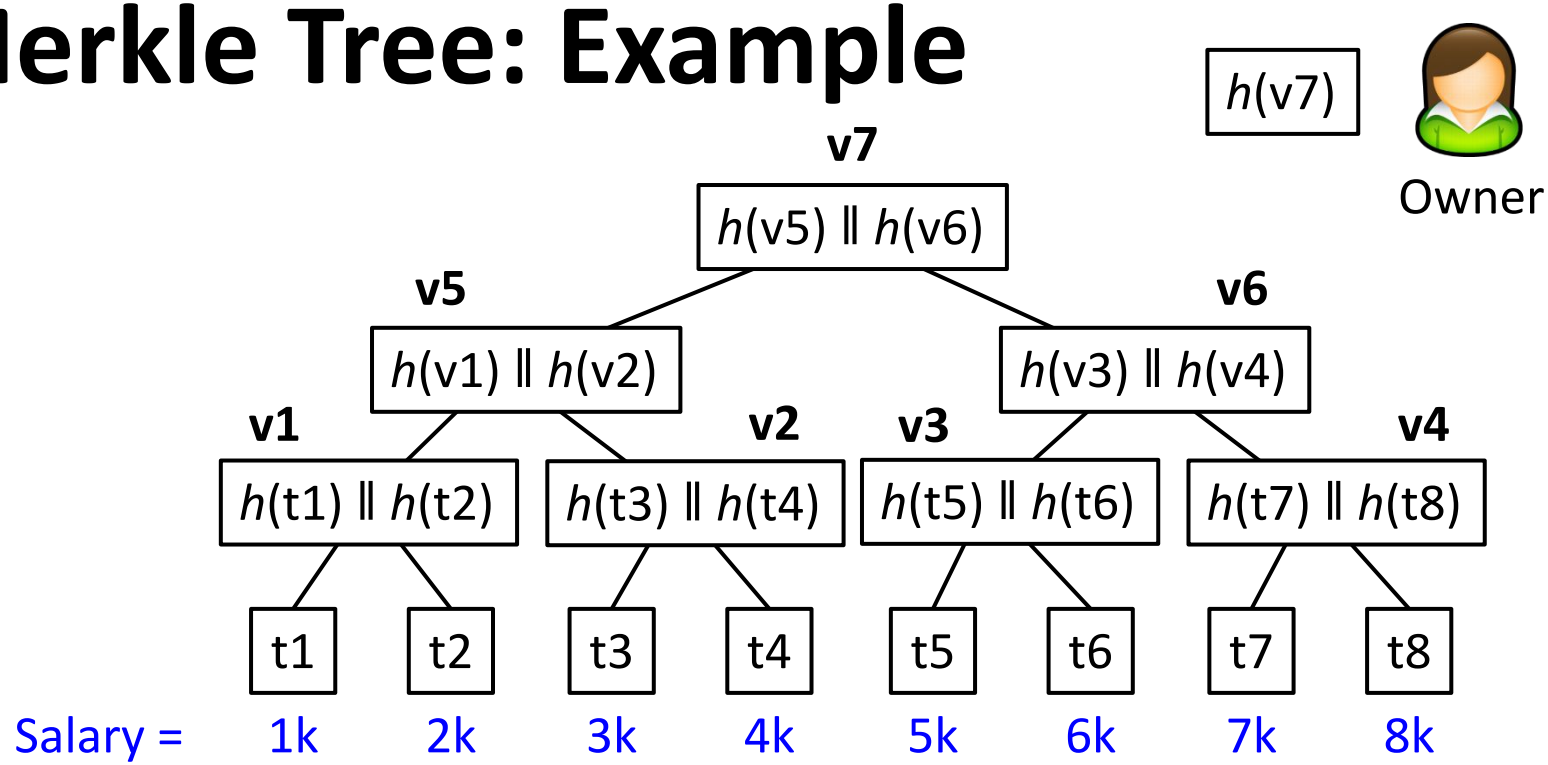
- Option 1: The user computes
  - *h*(t1), *h*(t2), *h*(t3), *h*(t4),
  - and then *h*(v1) ‖ *h*(v2)
  - and then verify it again *h*(v5)
- Question: does the user really need t1 and t2?
  - No; She only needs *h*(v1)
- That is, given t3, t4, and *h*(v1), the user can already verify the query result against *h*(v5)

Merkle Tree: Example



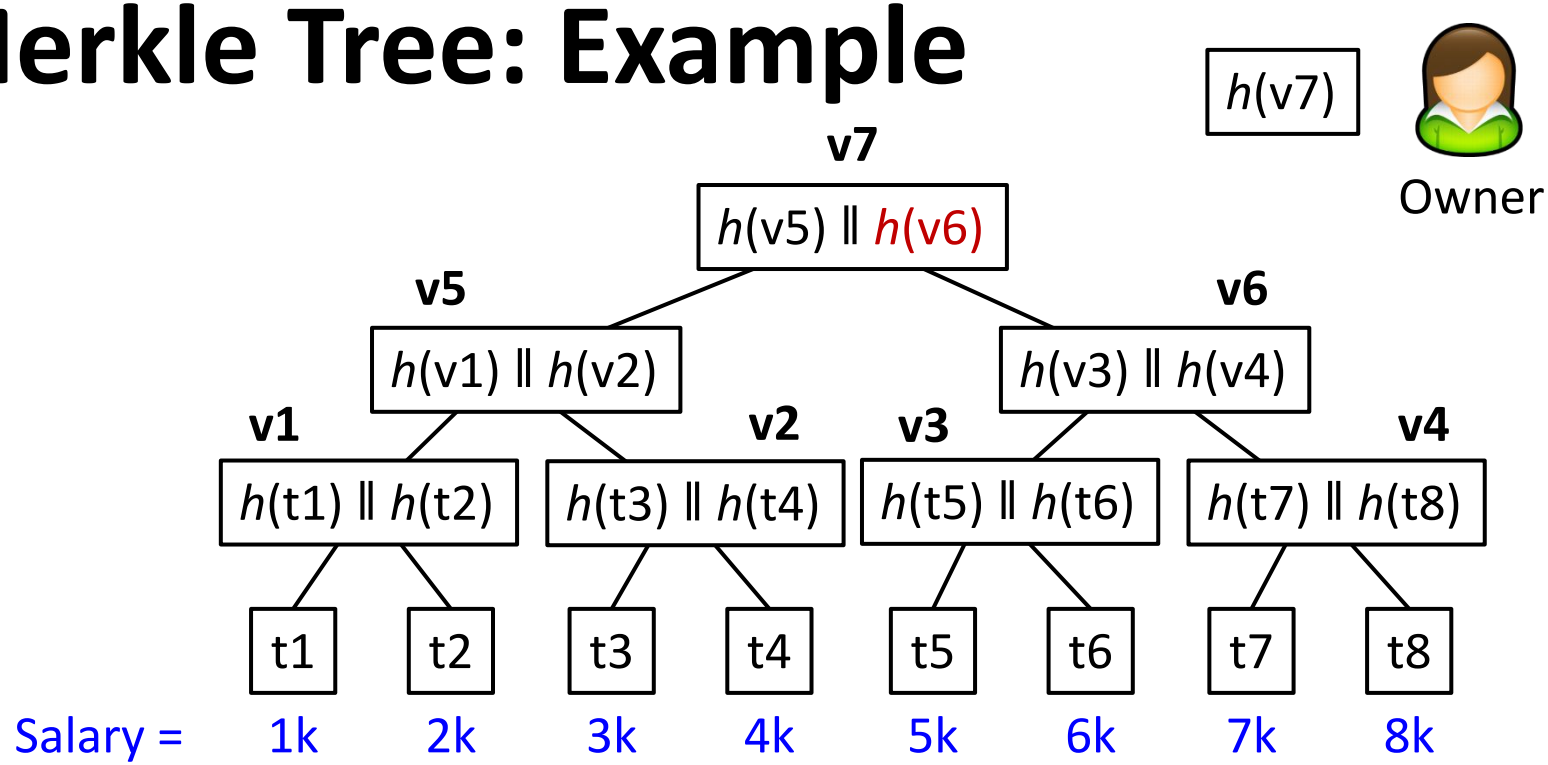
- So the service provider could simply return t3, t4, and *h*(v1), as well as the encrypted *h*(v5)
- This is sufficient for the user to verify the answer for "Salary = 3.5k"

Merkle Tree: Example



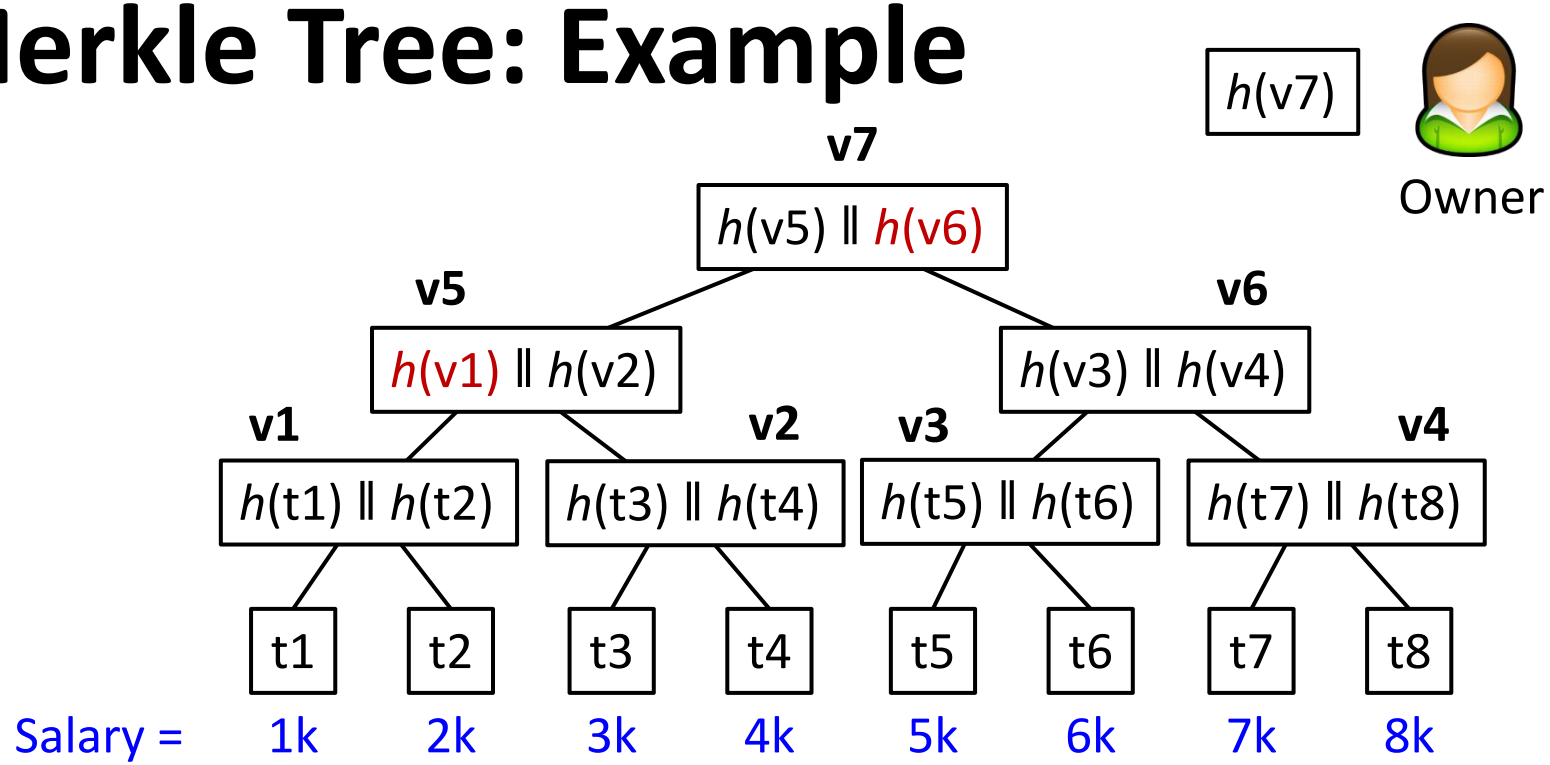
- Now reconsider the query on "Salary = 3.5k"
- The service provider could return the following:
  - t1, t2, t3, t4, t5, t6, t7, t8, and the encrypted *h*(v7)
- But t5, t6, t7, t8 could be replaced by *h*(v6)

Merkle Tree: Example



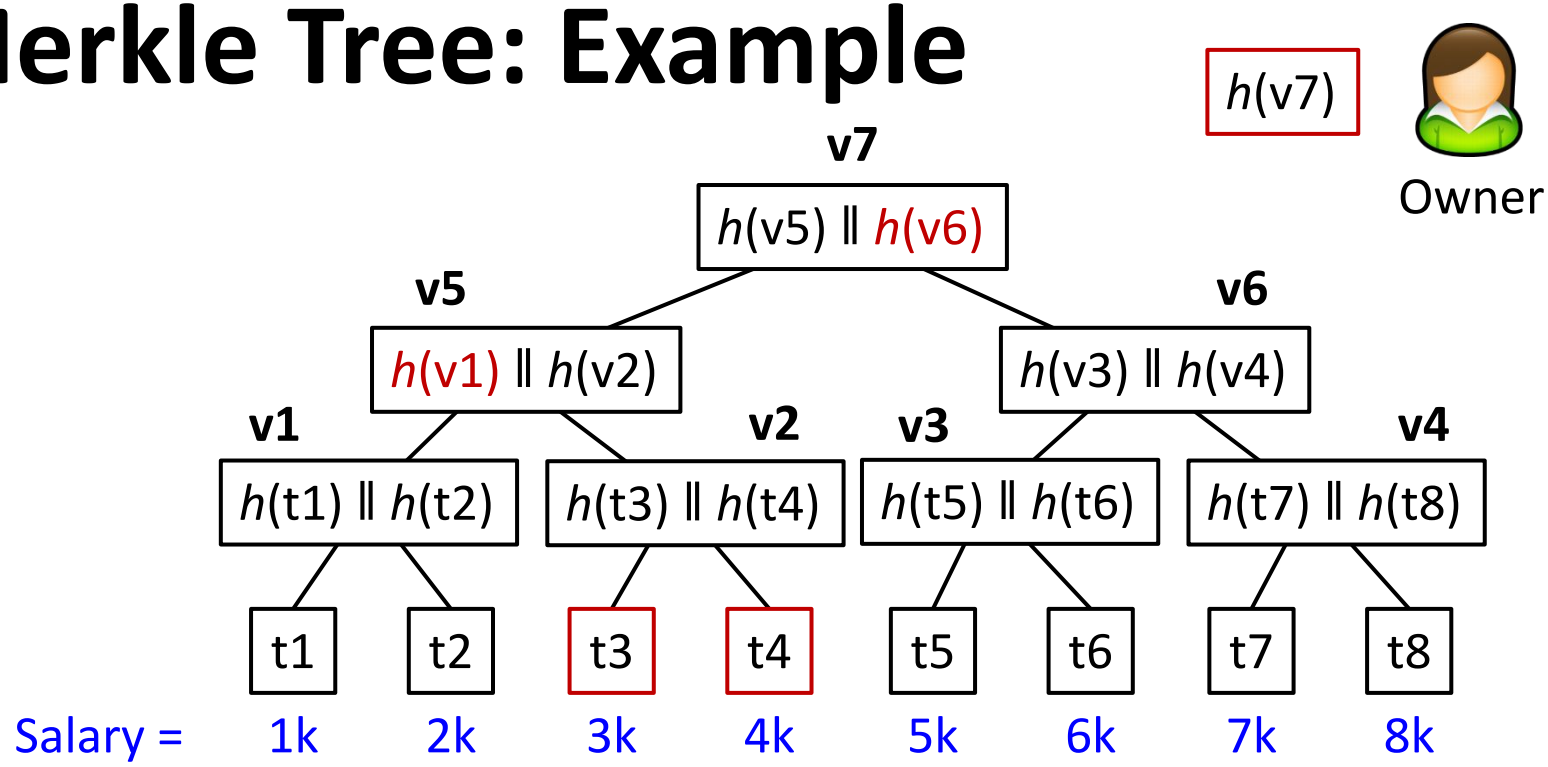
- Now reconsider the query on "Salary = 3.5k"
- The service provider could return the following:
  - t1, t2, t3, t4, *h*(v6), and the encrypted *h*(v7)
- But t1, t2 could be replaced by *h*(v1)

Merkle Tree: Example



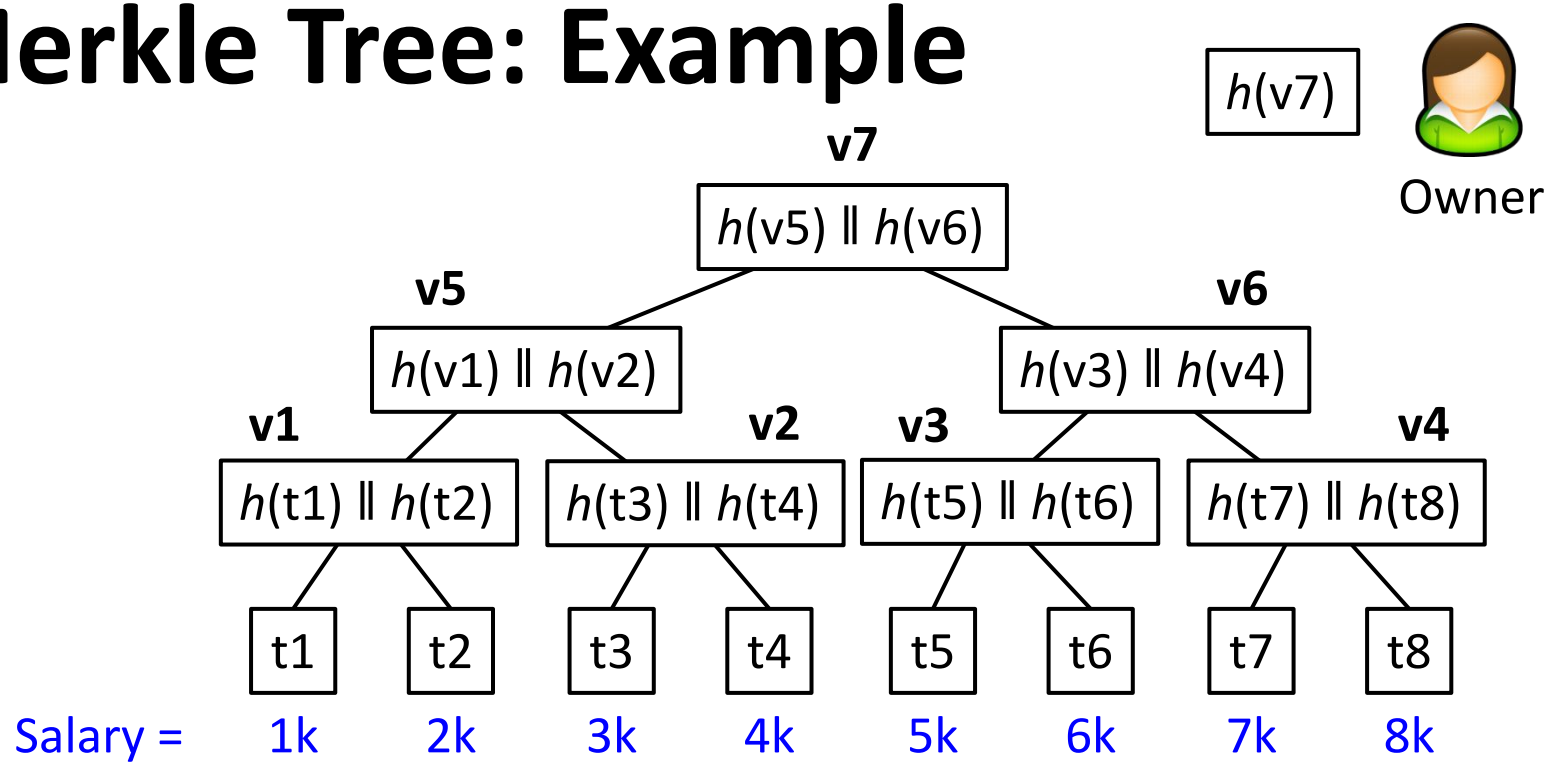
- Now reconsider the query on "Salary = 3.5k"
- The service provider could return the following:
  - *h*(v1), t3, t4, *h*(v6), and the encrypted *h*(v7)
- Any more replacement?
  - No; we definitely need t3, t4, and the encrypted *h*(v7)

Merkle Tree: Example



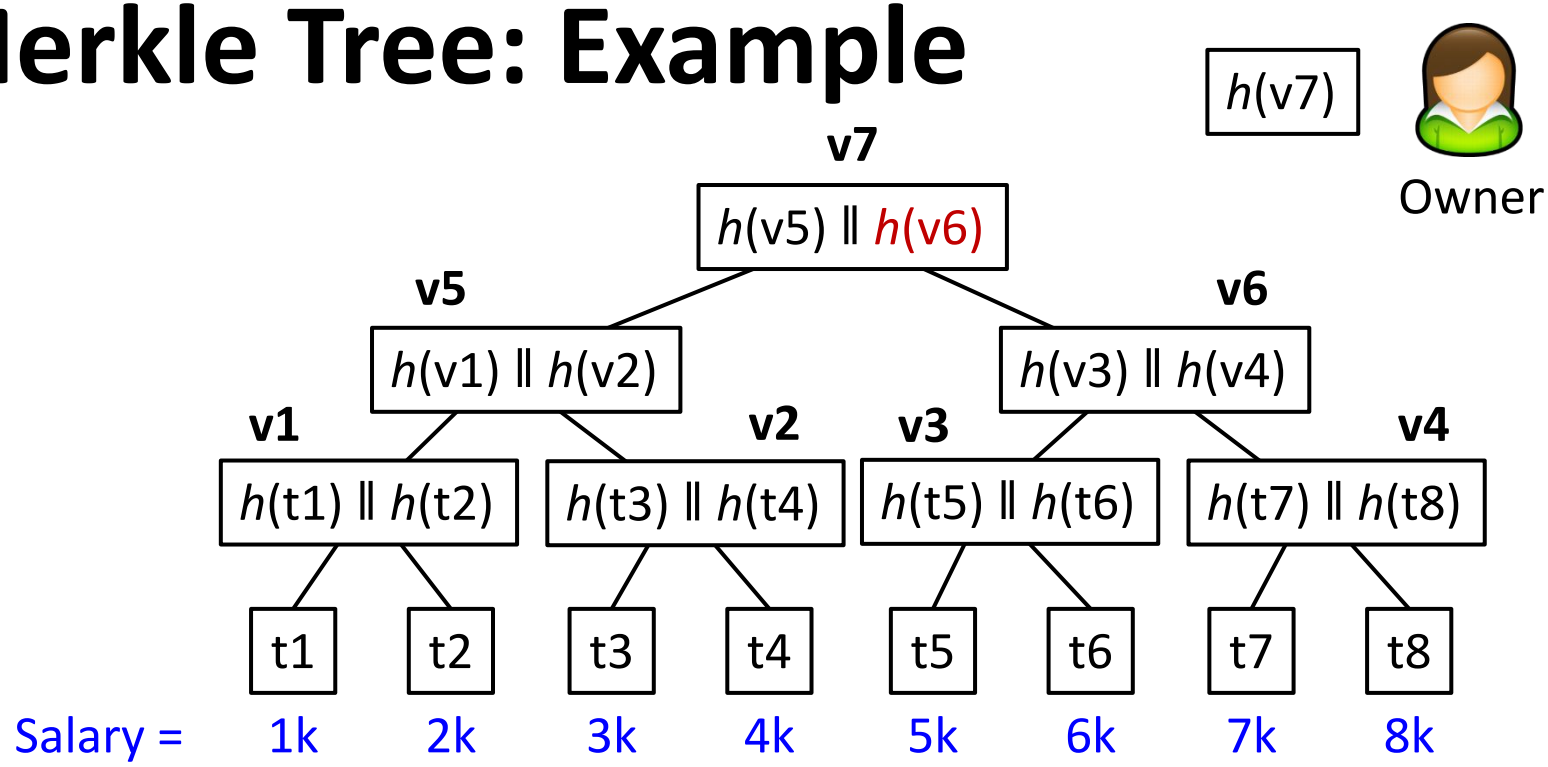
- Now reconsider the query on "Salary = 3.5k"
- Final answer: The service provider returns the following:
  - *h*(v1), t3, t4, *h*(v6), and the encrypted *h*(v7)

Merkle Tree: Example



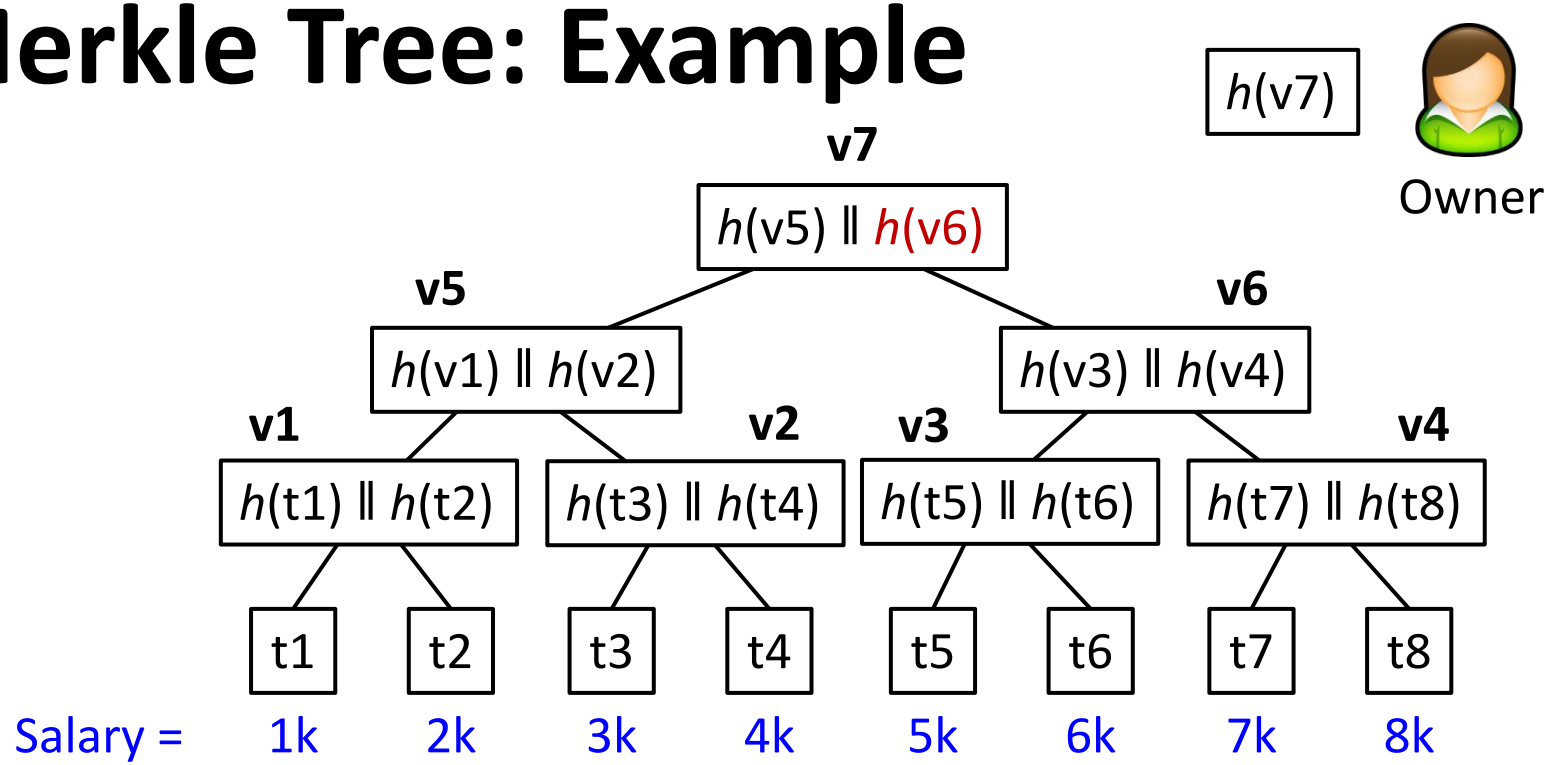
- Now consider a query on "Salary > 2.5k and Salary < 3.5k"
- The service provider could return the following:
  - t1, t2, t3, t4, t5, t6, t7, t8, and the encrypted *h*(v7)
- Any replacement possible?
  - t5, t6, t7, t8 could be replaced by *h*(v6)

Merkle Tree: Example



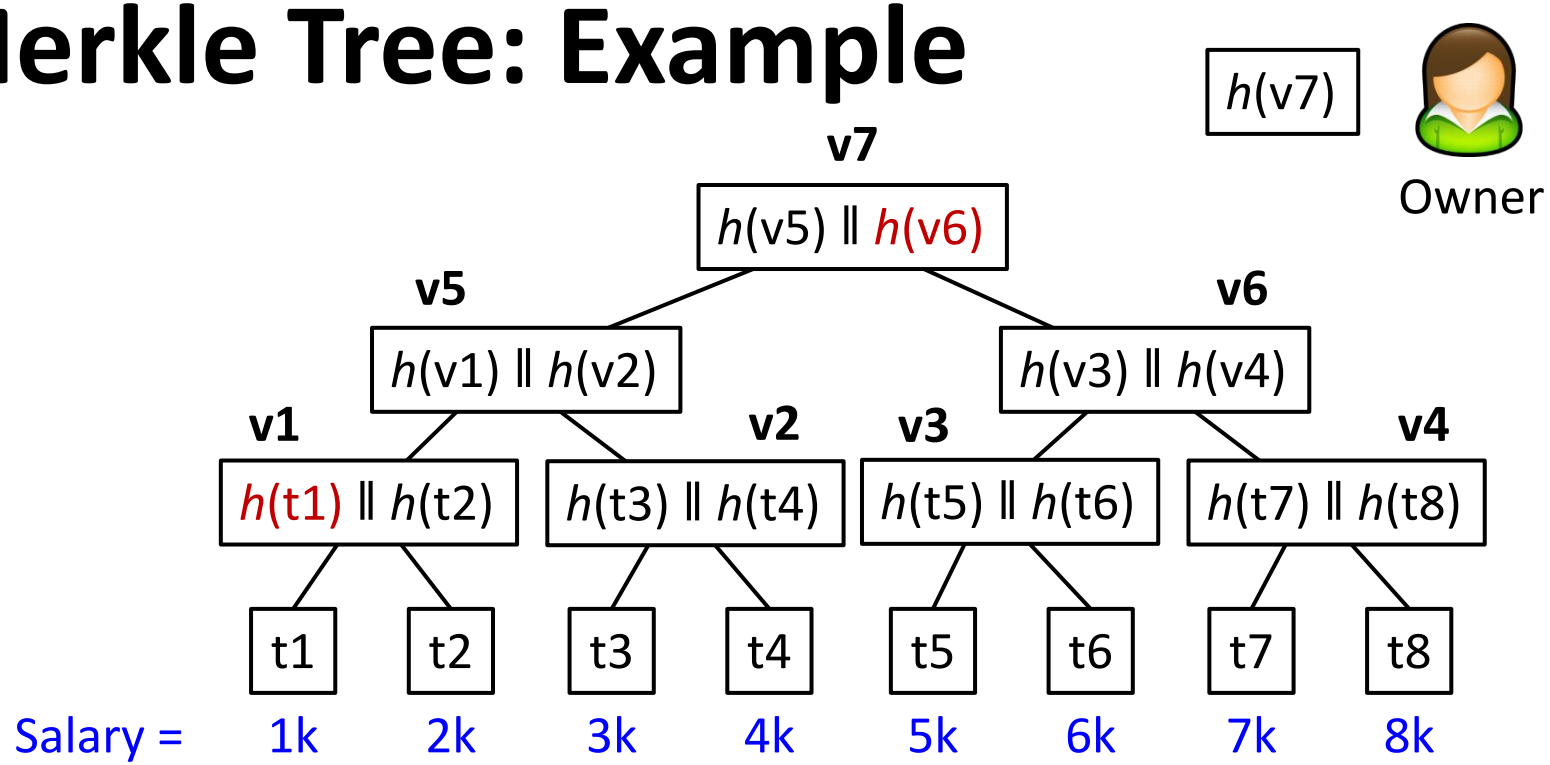
- Now consider a query on "Salary > 2.5k and Salary < 3.5k"
- The service provider could return the following:
  - t1, t2, t3, t4, *h*(v6), and the encrypted *h*(v7)
- Could we replace t1, t2 with *h*(v1)?
  - No; otherwise, the user cannot verify whether the service provider has hidden a tuple with Salary = 2.6k

Merkle Tree: Example



- Now consider a query on "Salary > 2.5k and Salary < 3.5k"
- The service provider could return the following:
  - t1, t2, t3, t4, *h*(v6), and the encrypted *h*(v7)
- Could we replace t1 with *h*(t1)?
  - This is OK

Merkle Tree: Example

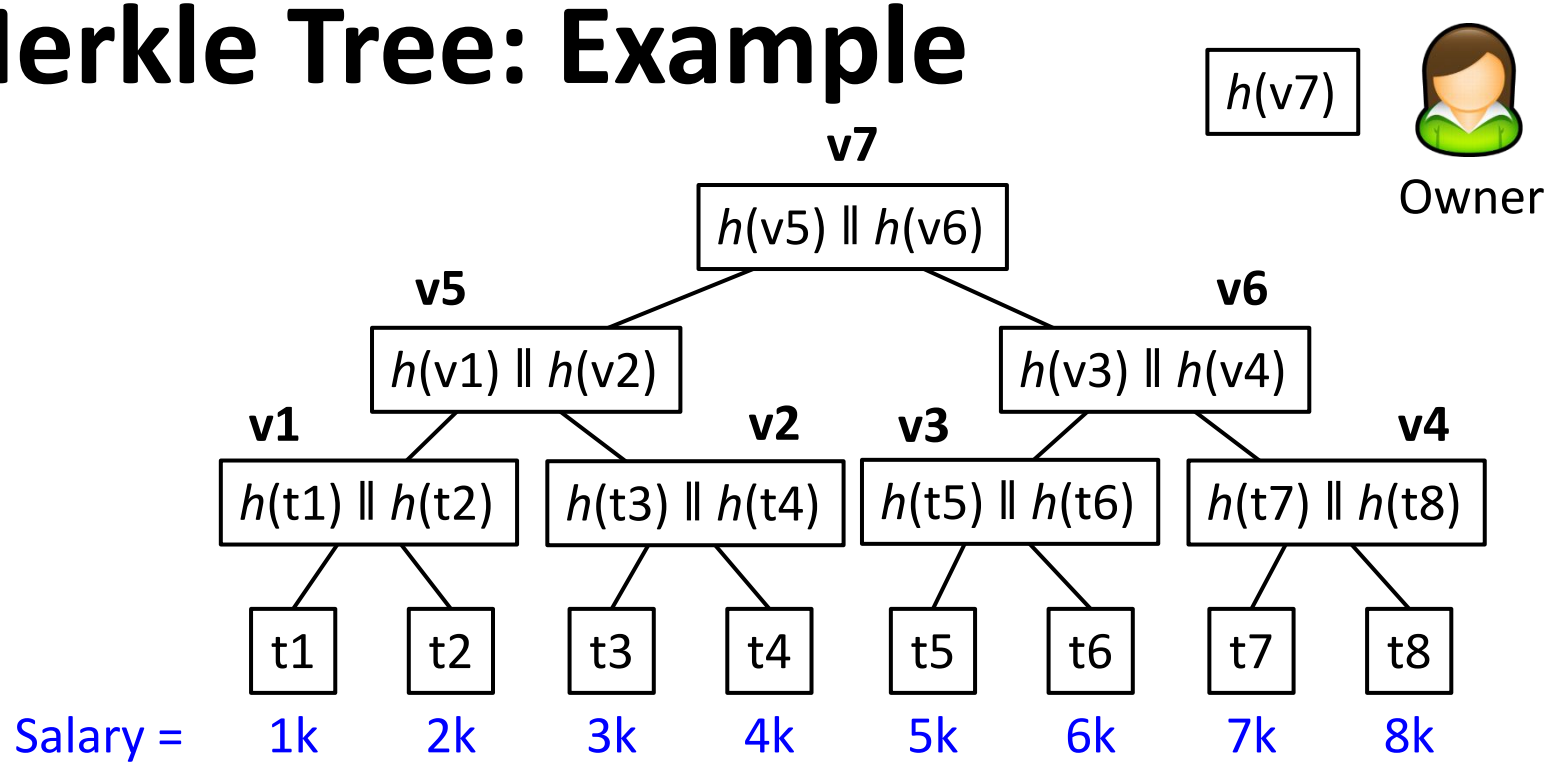


- Now consider a query on "Salary > 2.5k and Salary < 3.5k"
- The service provider could return the following:
  - *h*(t1), t2, t3, t4, *h*(v6), and the encrypted *h*(v7)
- Any more replacement?
  - No; we definitely need t2, t3, t4 to prove correctness

Merkle Tree: General Algorithm

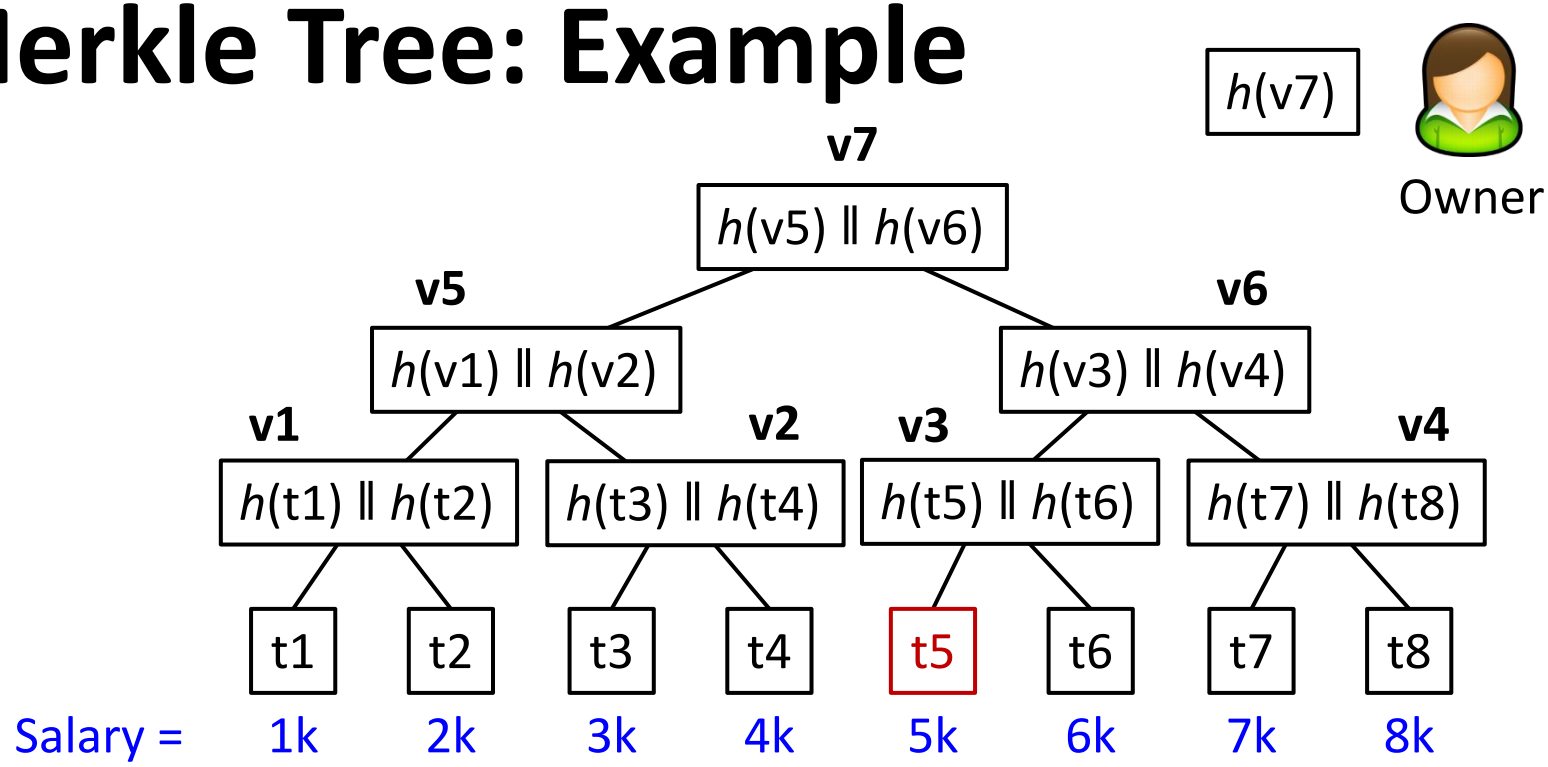
- Consider a query on T.A in [x, y]
- Among the tuples t with t.A < x, find the tuple tx whose A value is the largest
  - Identify the path from tx to the root of the Merkel tree
  - For every “left branch” on the path, collect the hash value of the branch
- Among the tuples t with t.A > y, find the tuple ty whose A value is the smallest
  - Identify the path from ty to the root of the Merkel tree
  - For every “right branch” on the path, collect the hash value of the branch
- Return tx, ty, and all tuples between them, and all hash values collected, as well as the encrypted Merkle root

Merkle Tree: Example



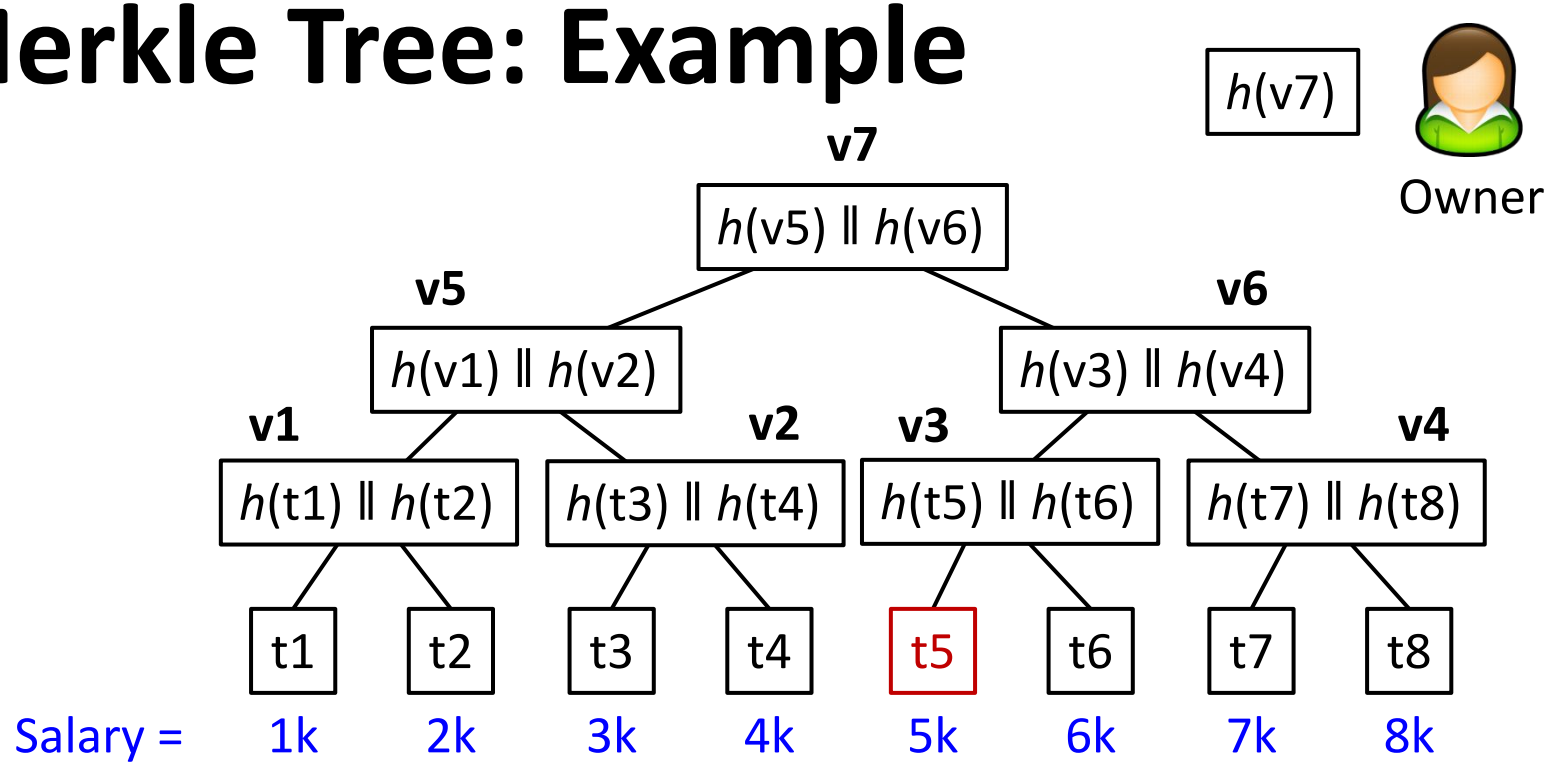
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Among the tuples t with t.A < x, find the tuple tx whose A value is the largest"
  - tx is t5

Merkle Tree: Example



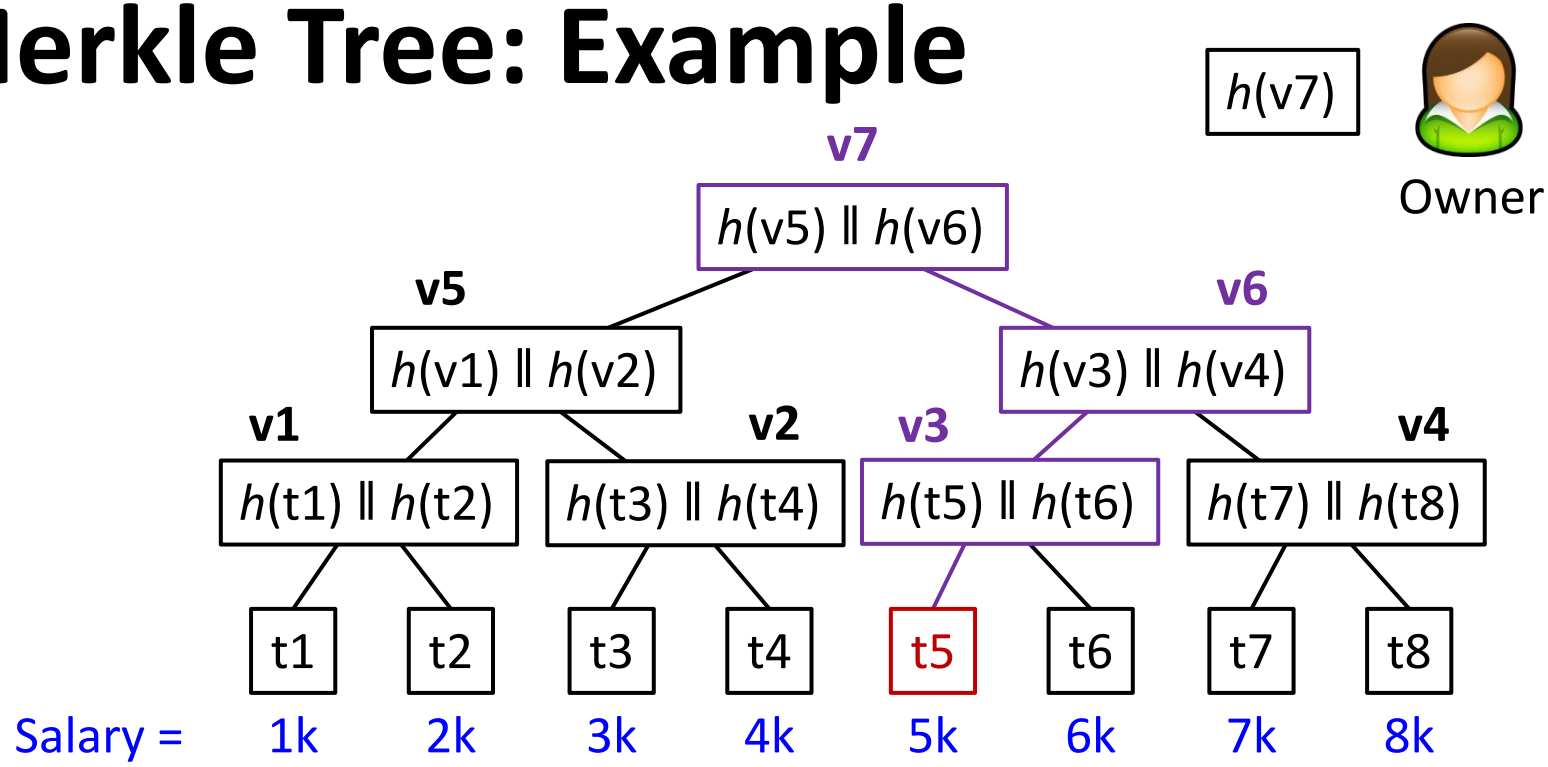
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Among the tuples t with t.A < x, find the tuple tx whose A value is the largest"
  - tx is t5

Merkle Tree: Example



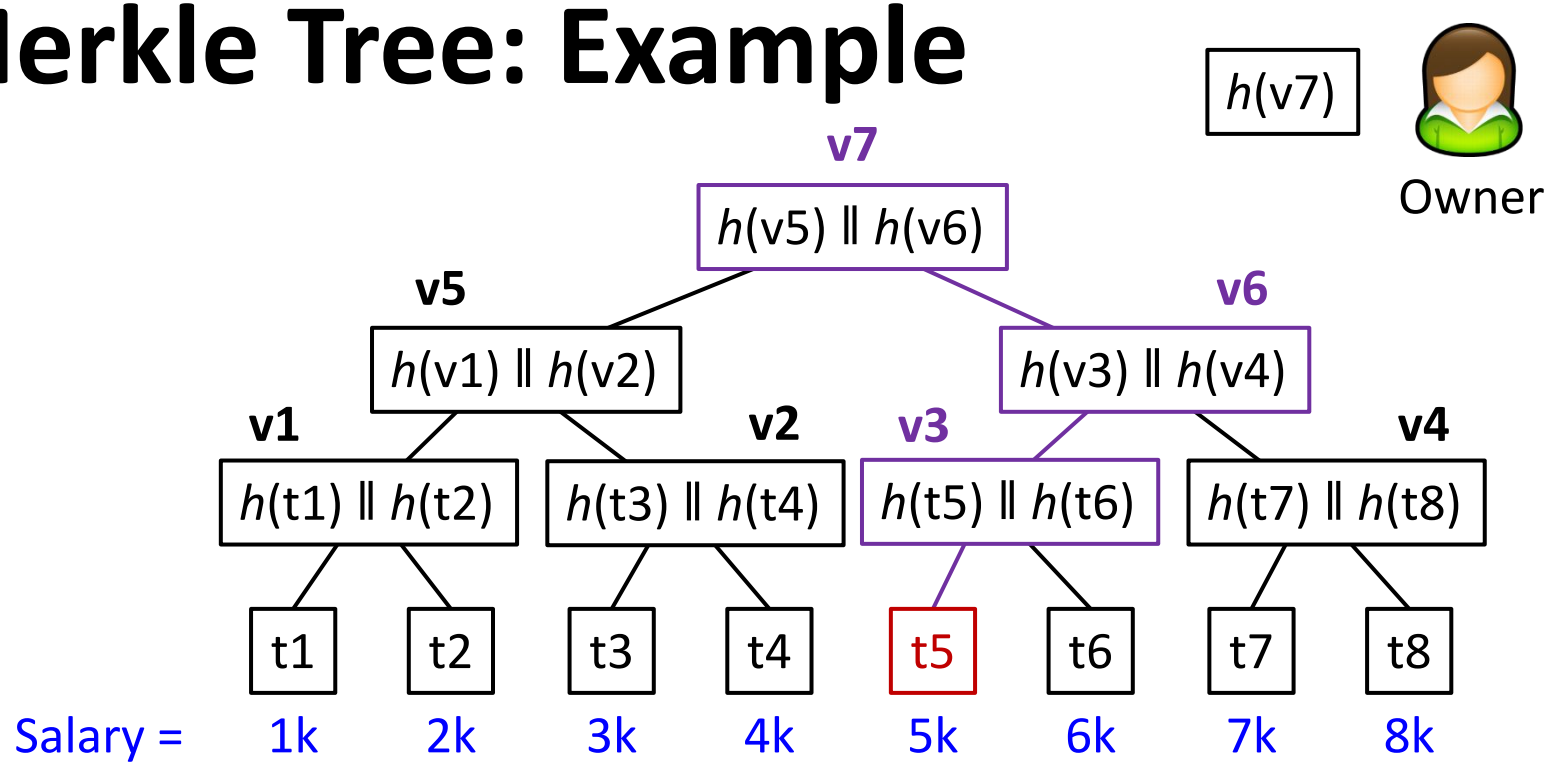
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Identify the path from tx to the root of the Merkel tree"
  - The path from t5 to v7

Merkle Tree: Example



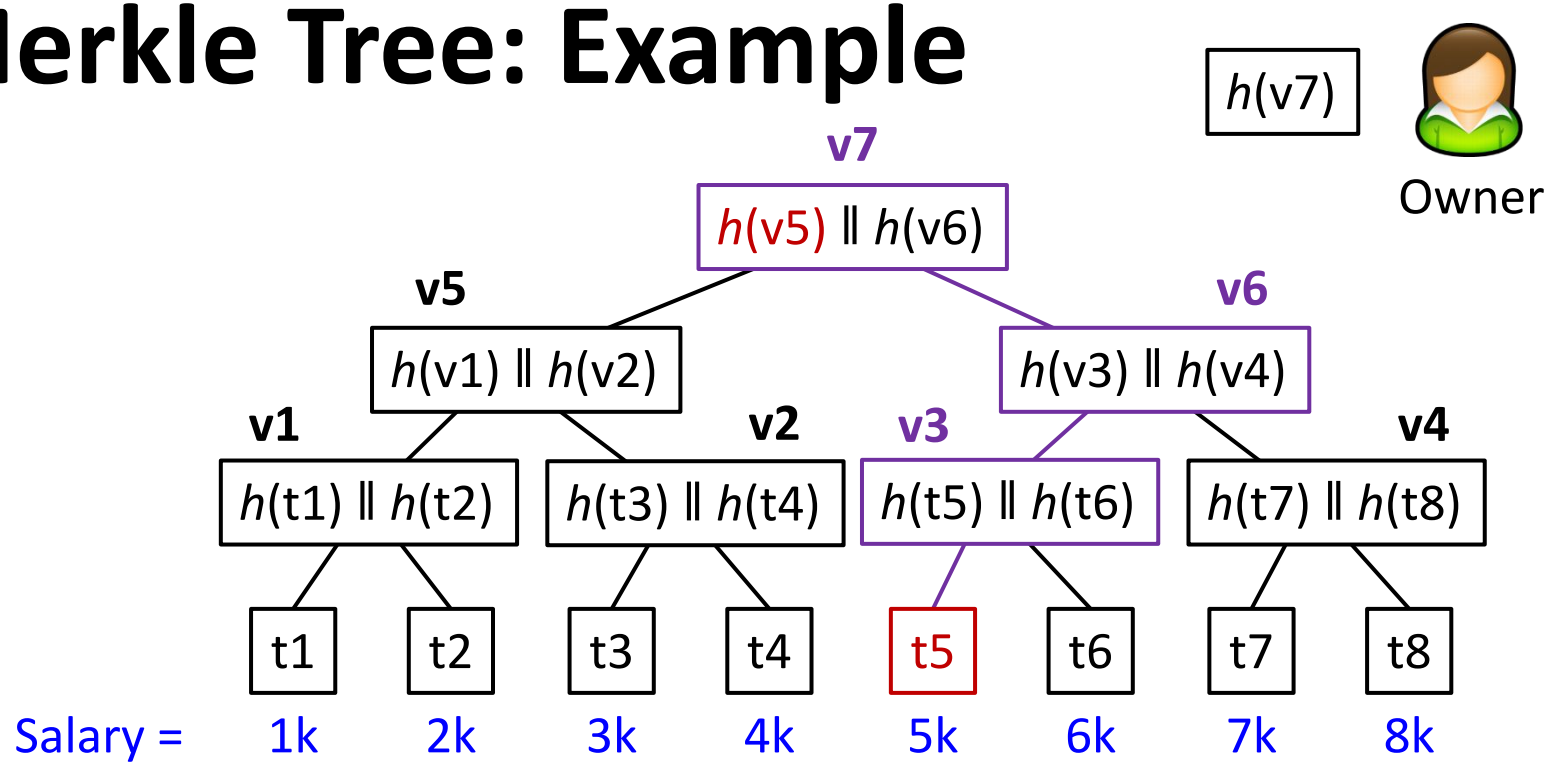
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Identify the path from tx to the root of the Merkel tree"
  - The path from t5 to v7

Merkle Tree: Example



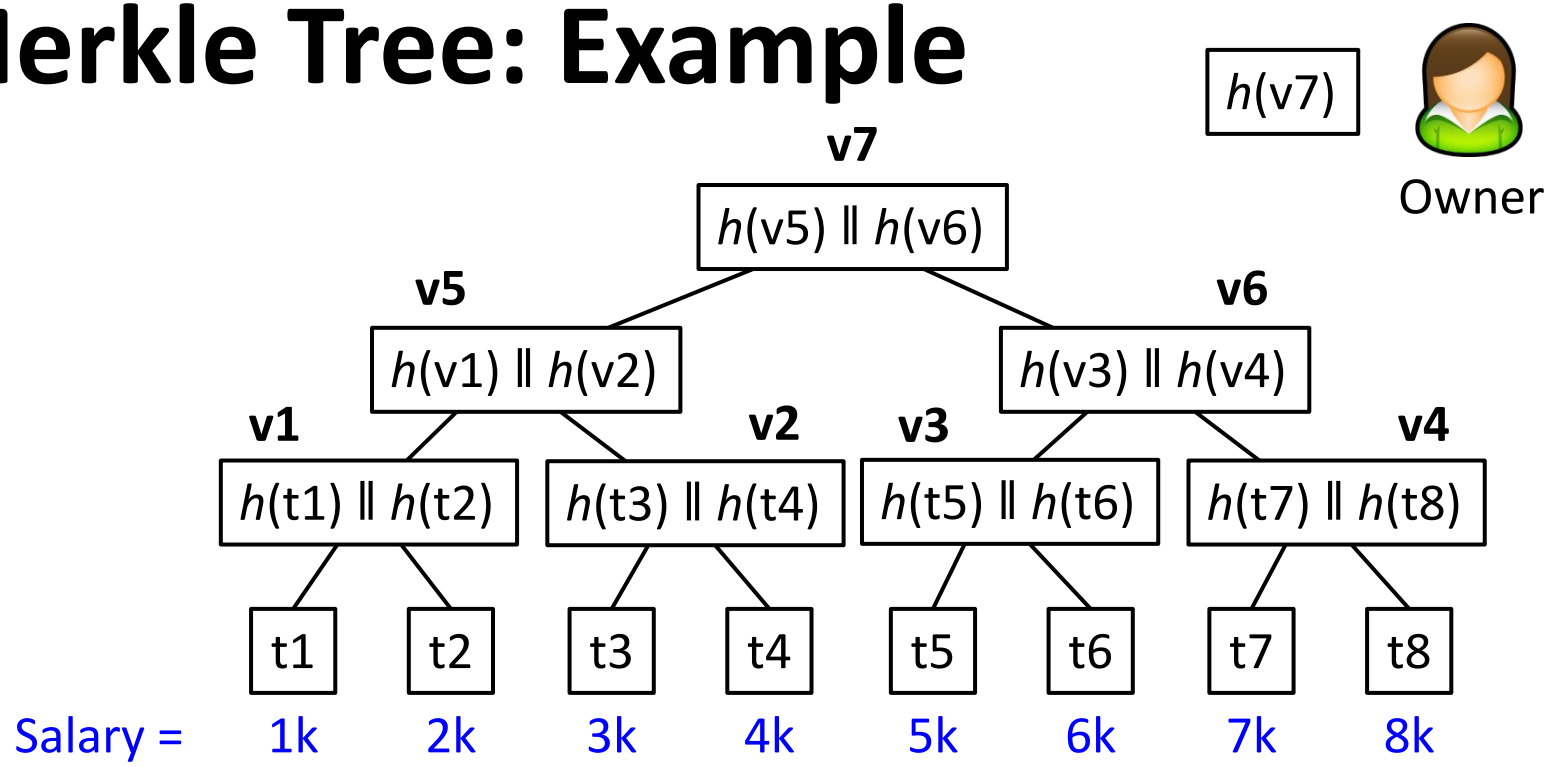
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "For every left branch on the path, collect the hash value of the branch"
  - Collected hash: *h*(v5)

Merkle Tree: Example



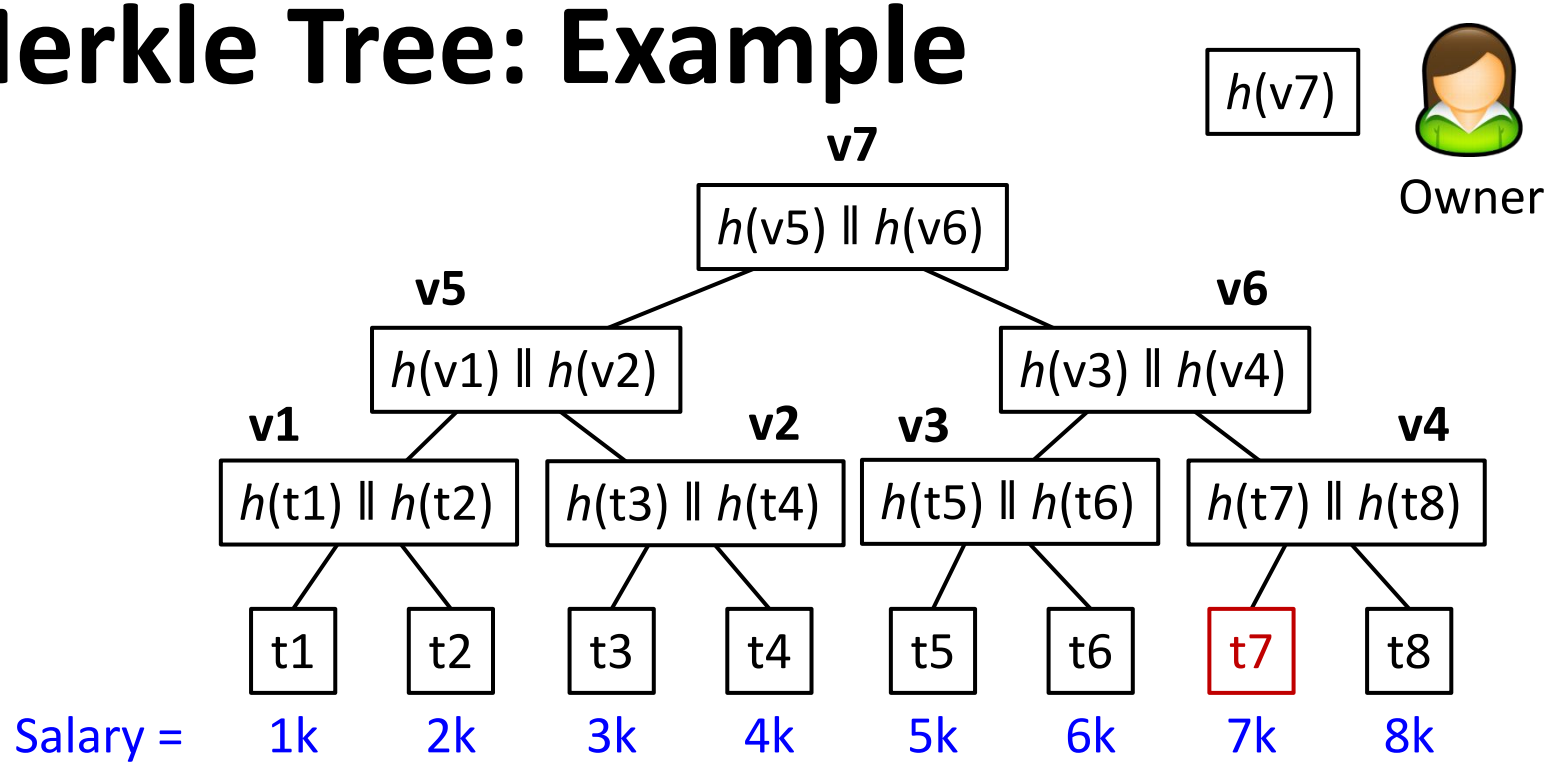
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "For every left branch on the path, collect the hash value of the branch"
  - Collected hash: *h*(v5)

Merkle Tree: Example



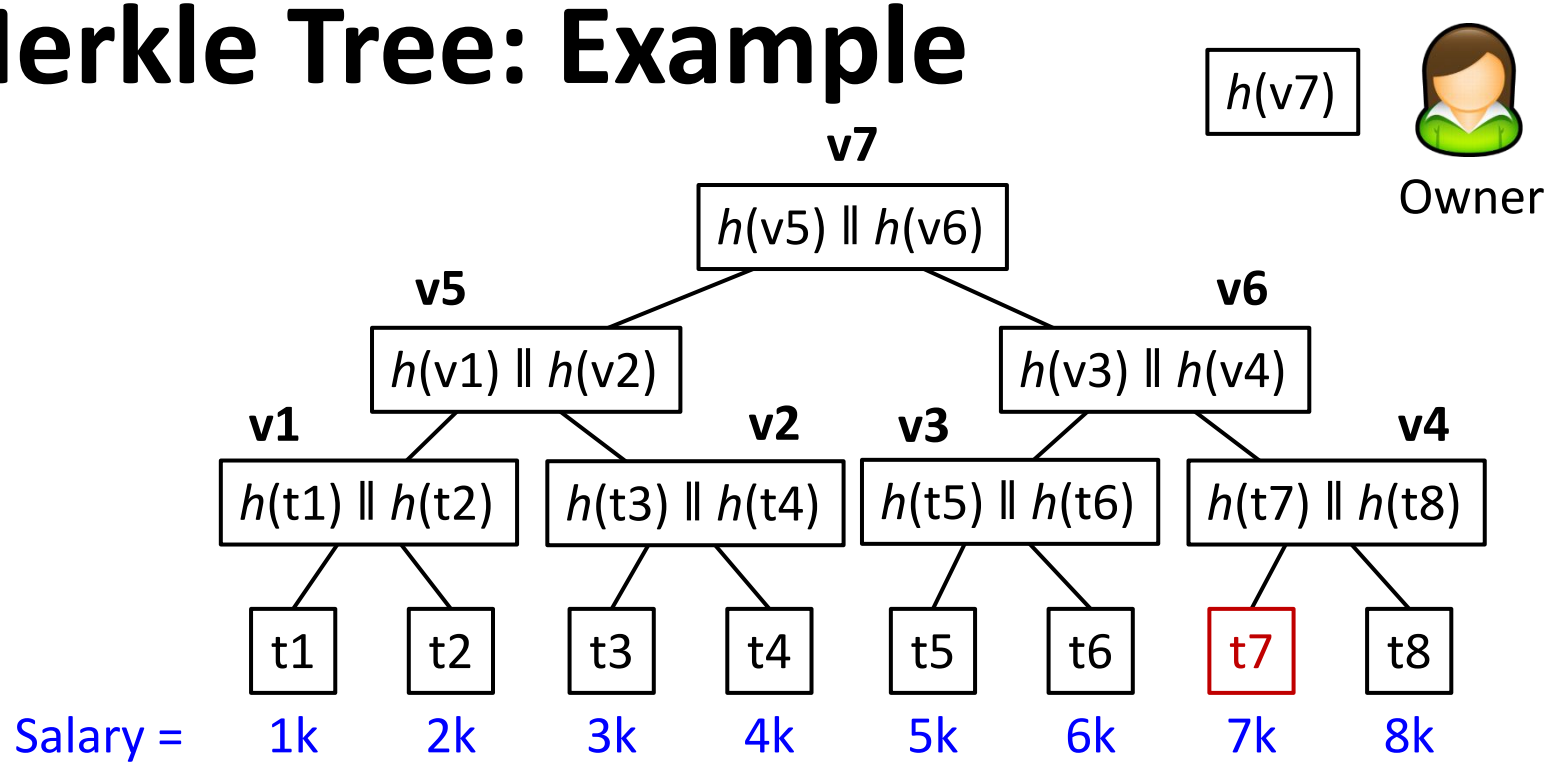
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Among the tuples t with t.A > y, find the tuple ty whose A value is the smallest"
  - ty is t7

Merkle Tree: Example



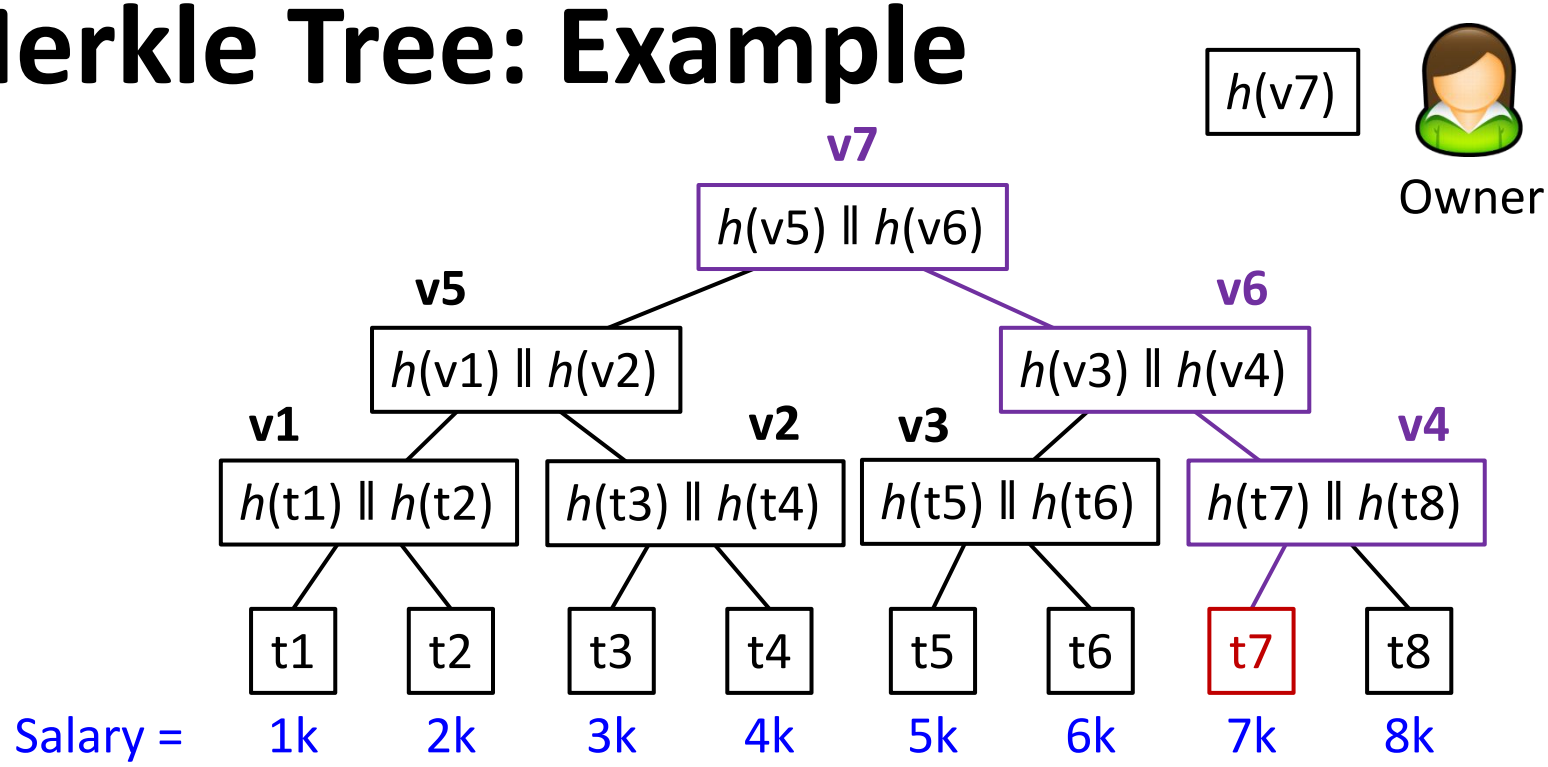
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Among the tuples t with t.A > y, find the tuple ty whose A value is the smallest"
  - ty is t7

Merkle Tree: Example



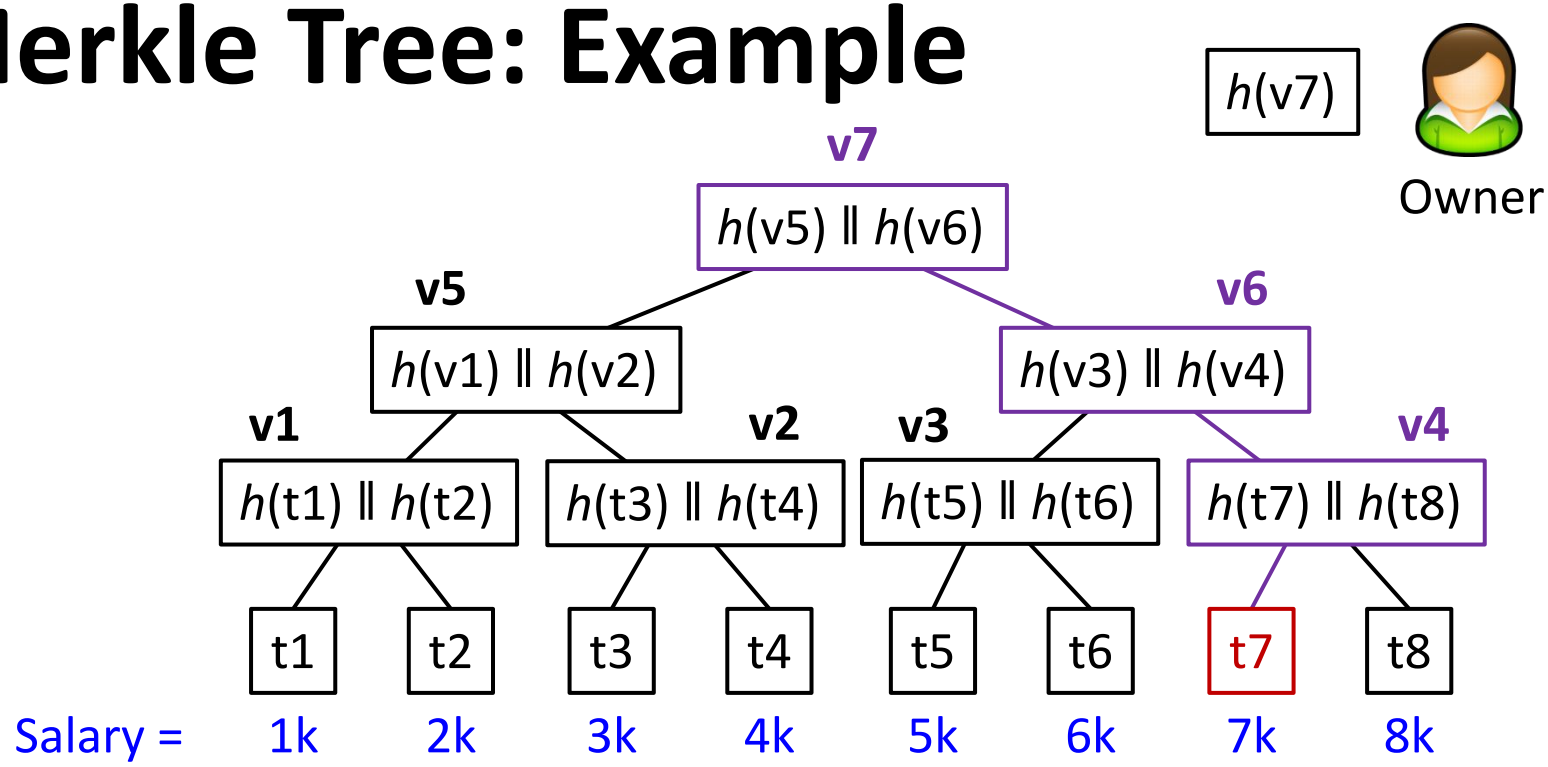
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Identify the path from ty to the root of the Merkel tree"
  - the path from t7 to v7

Merkle Tree: Example



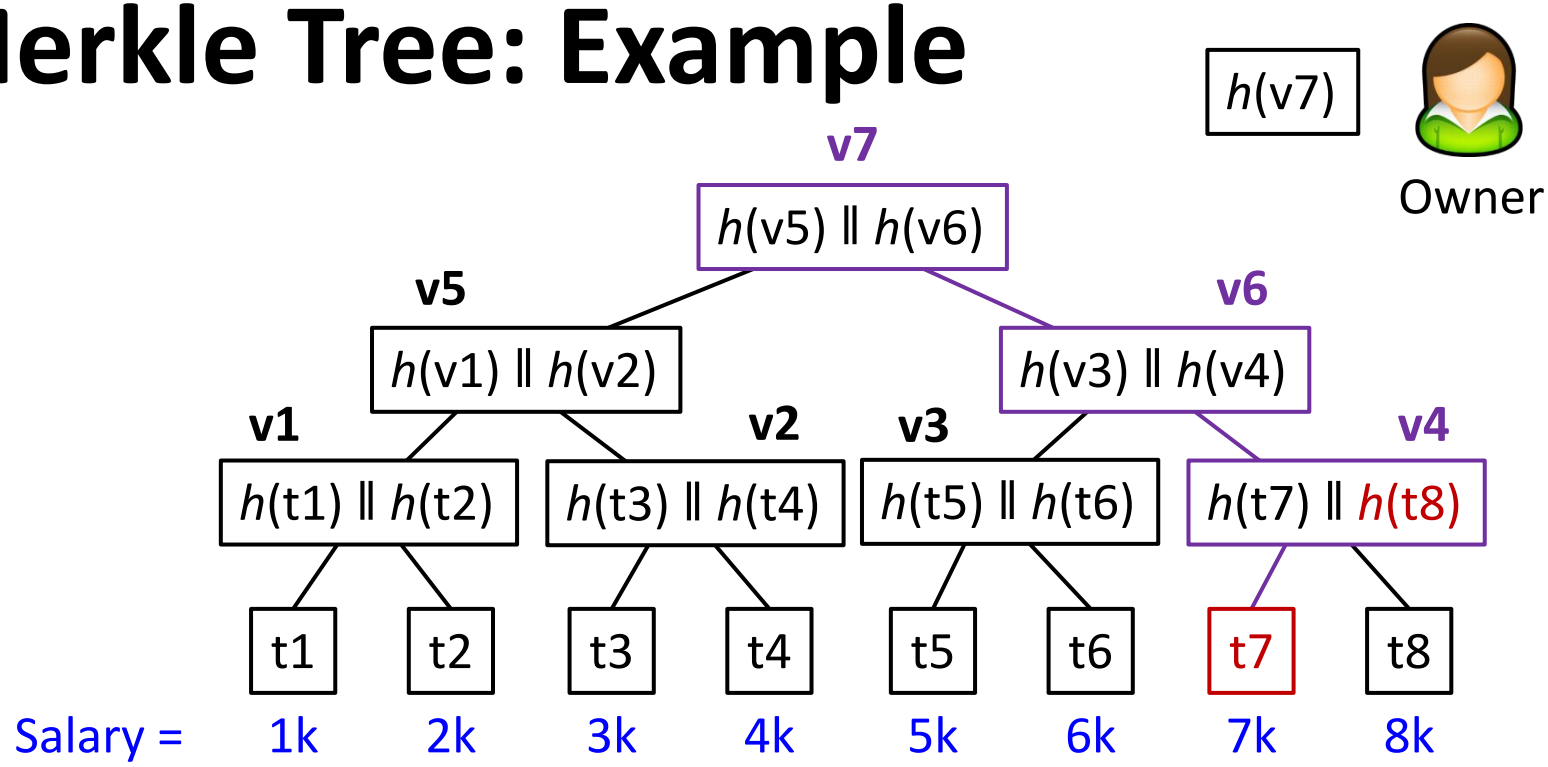
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Identify the path from ty to the root of the Merkel tree"
  - the path from t7 to v7

Merkle Tree: Example



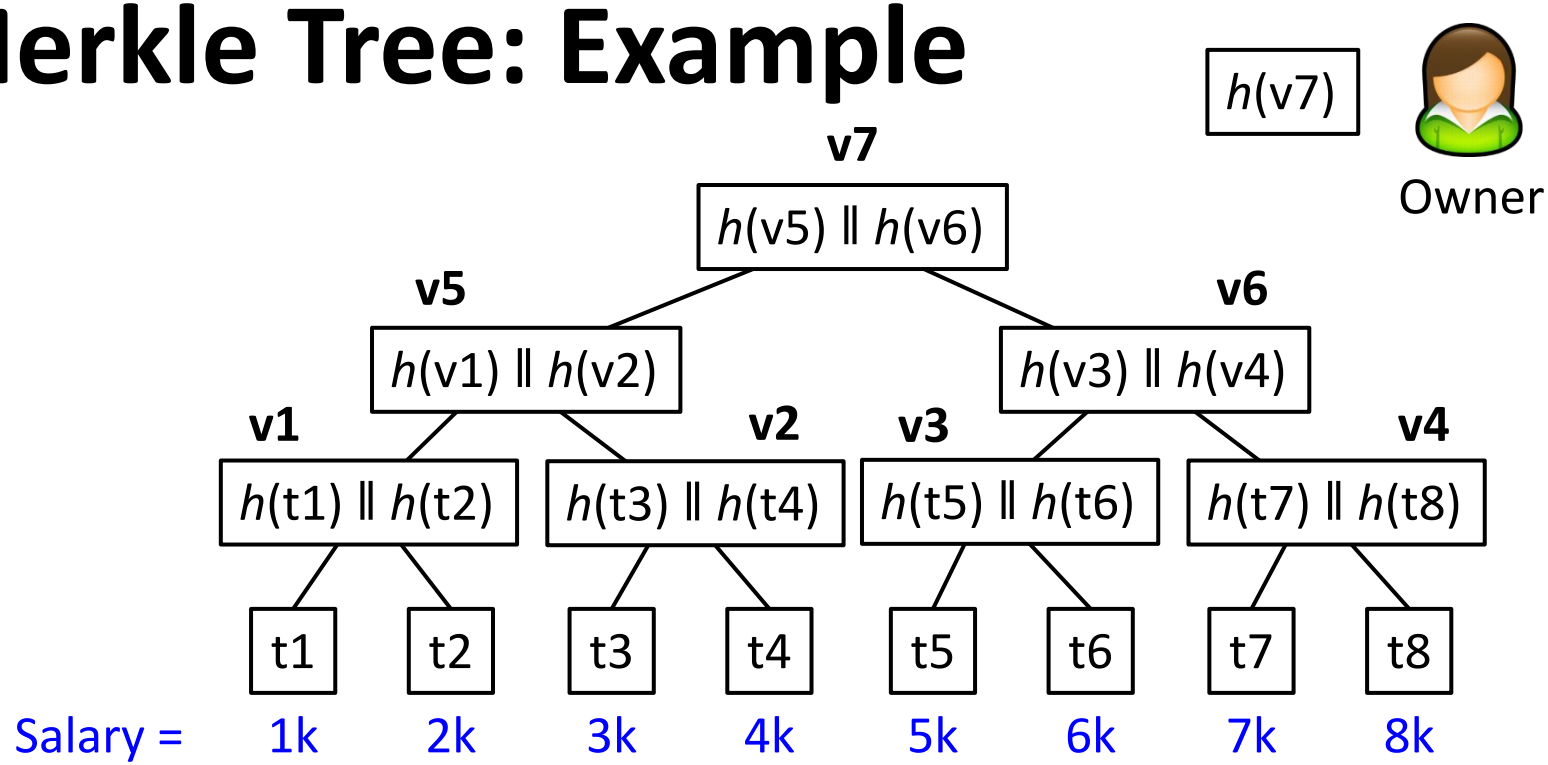
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "For every right branch on the path, collect the hash value of the branch"
  - Collected hashes: *h*(t8)

Merkle Tree: Example



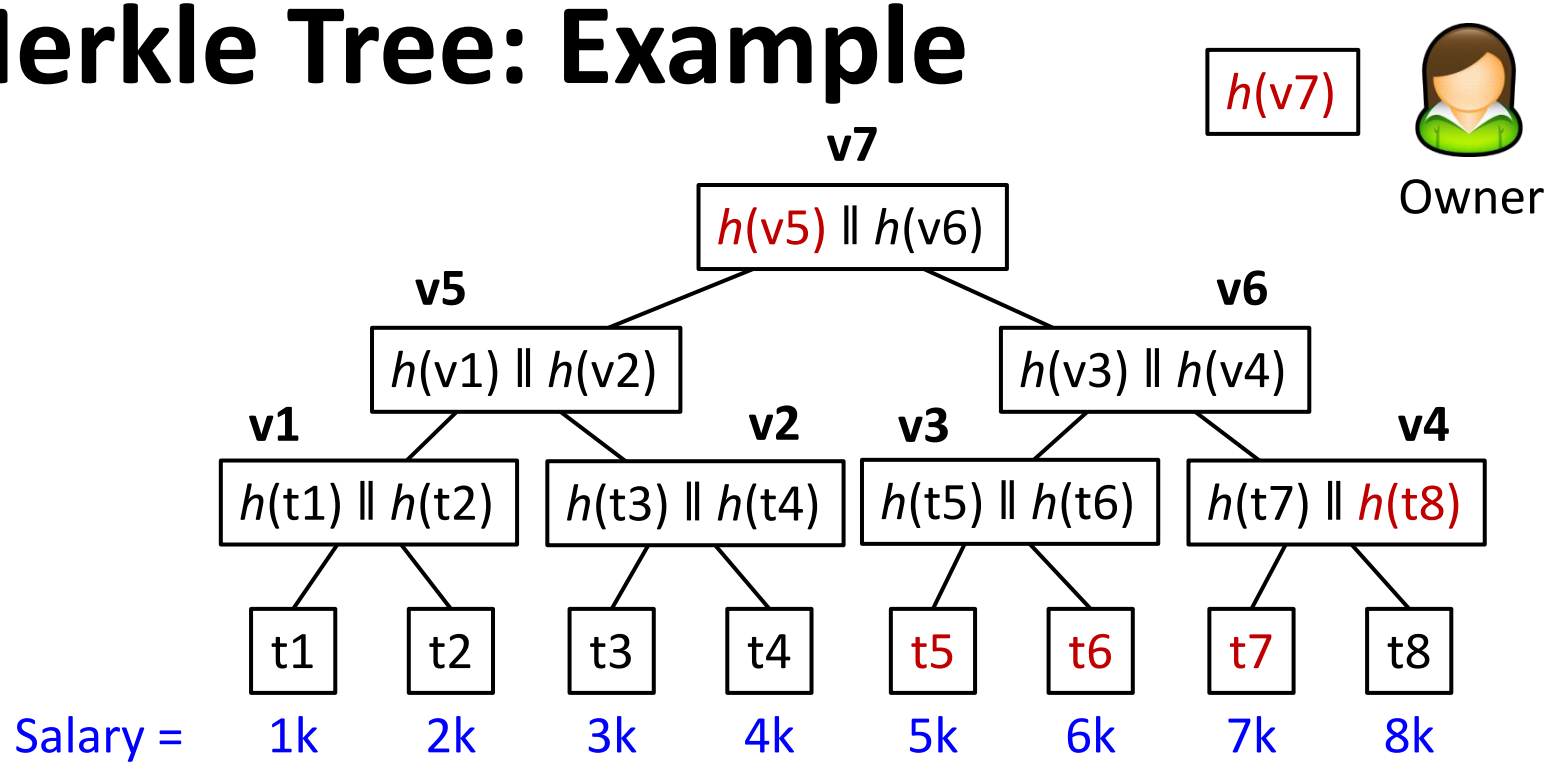
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "For every right branch on the path, collect the hash value of the branch"
  - Collected hashes: *h*(t8)

Merkle Tree: Example



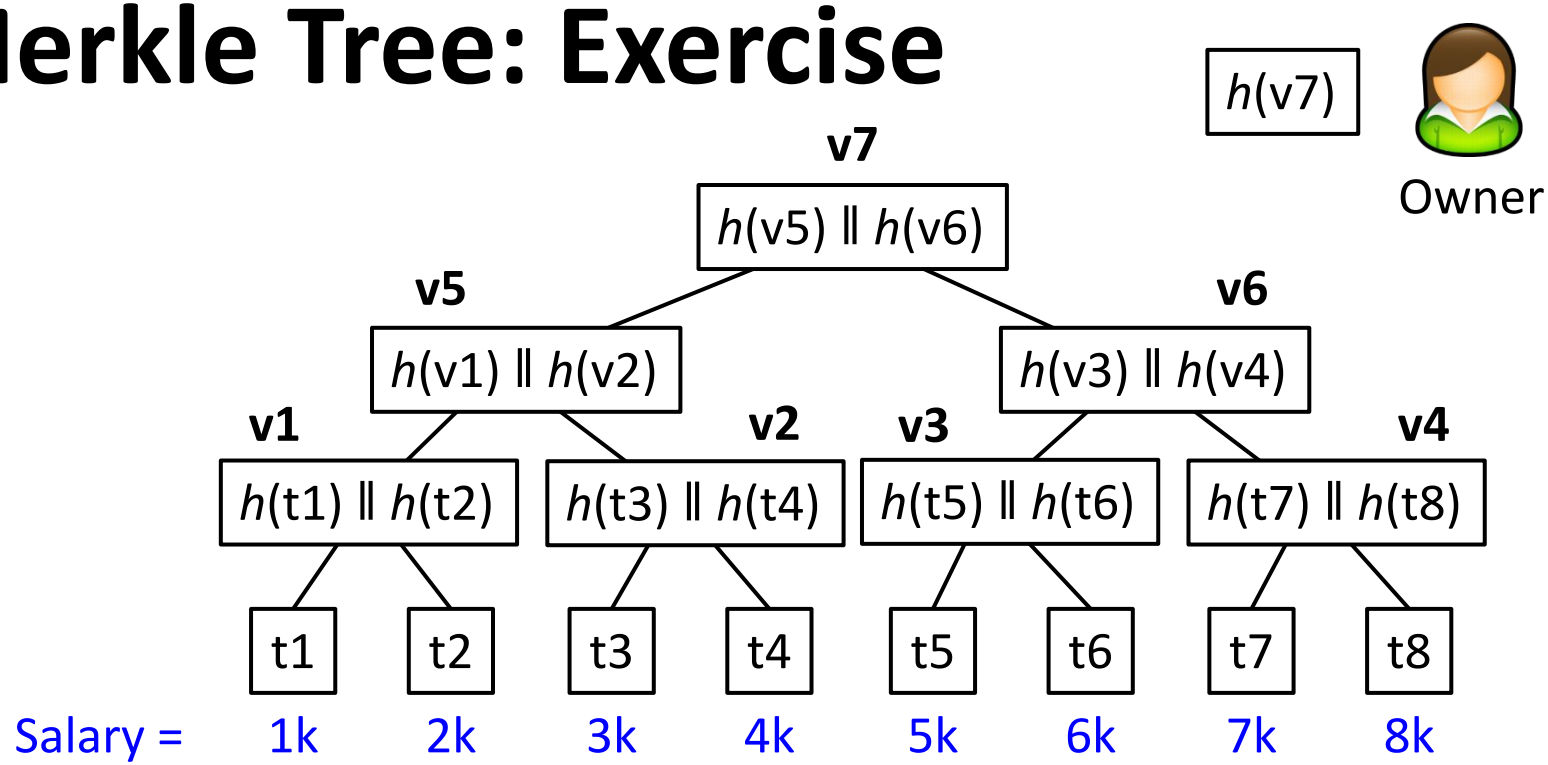
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Return tx, ty, and all tuples between them, and all hash values collected, as well as the encrypted Merkle root"
  - i.e., t5, t7, and t6, and *h*(v5) and *h*(t8), and the encrypted *h*(v7)

Merkle Tree: Example



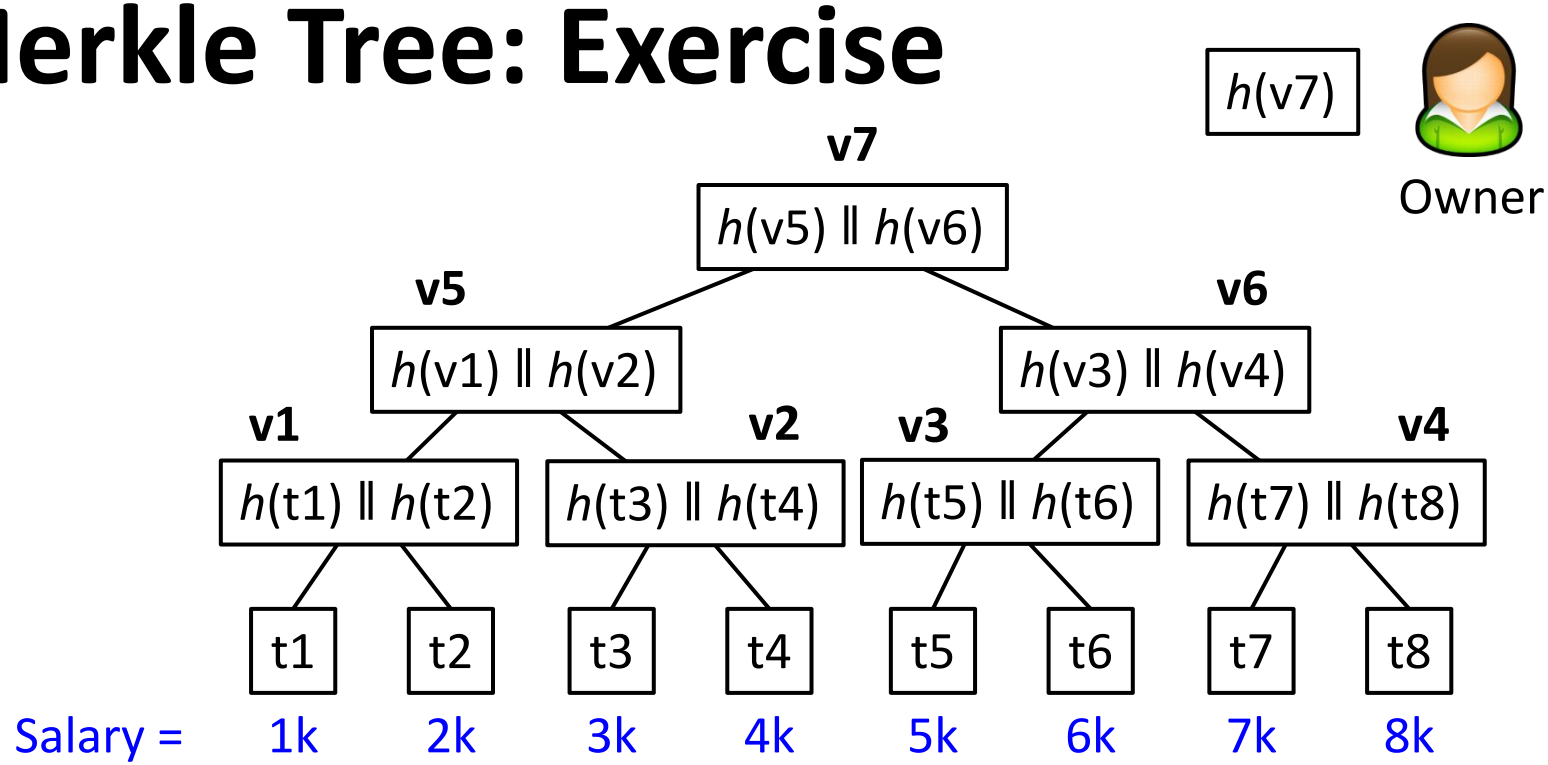
- Consider a query on "Salary in [5.5k, 6.5k]"
  - i.e., A is Salary, and [x, y] = [5.5k, 6.5k]
- "Return tx, ty, and all tuples between them, and all hash values collected, as well as the encrypted Merkle root"
  - i.e., t5, t7, and t6, and *h*(v5) and *h*(t8), and the encrypted *h*(v7)

Merkle Tree: Exercise



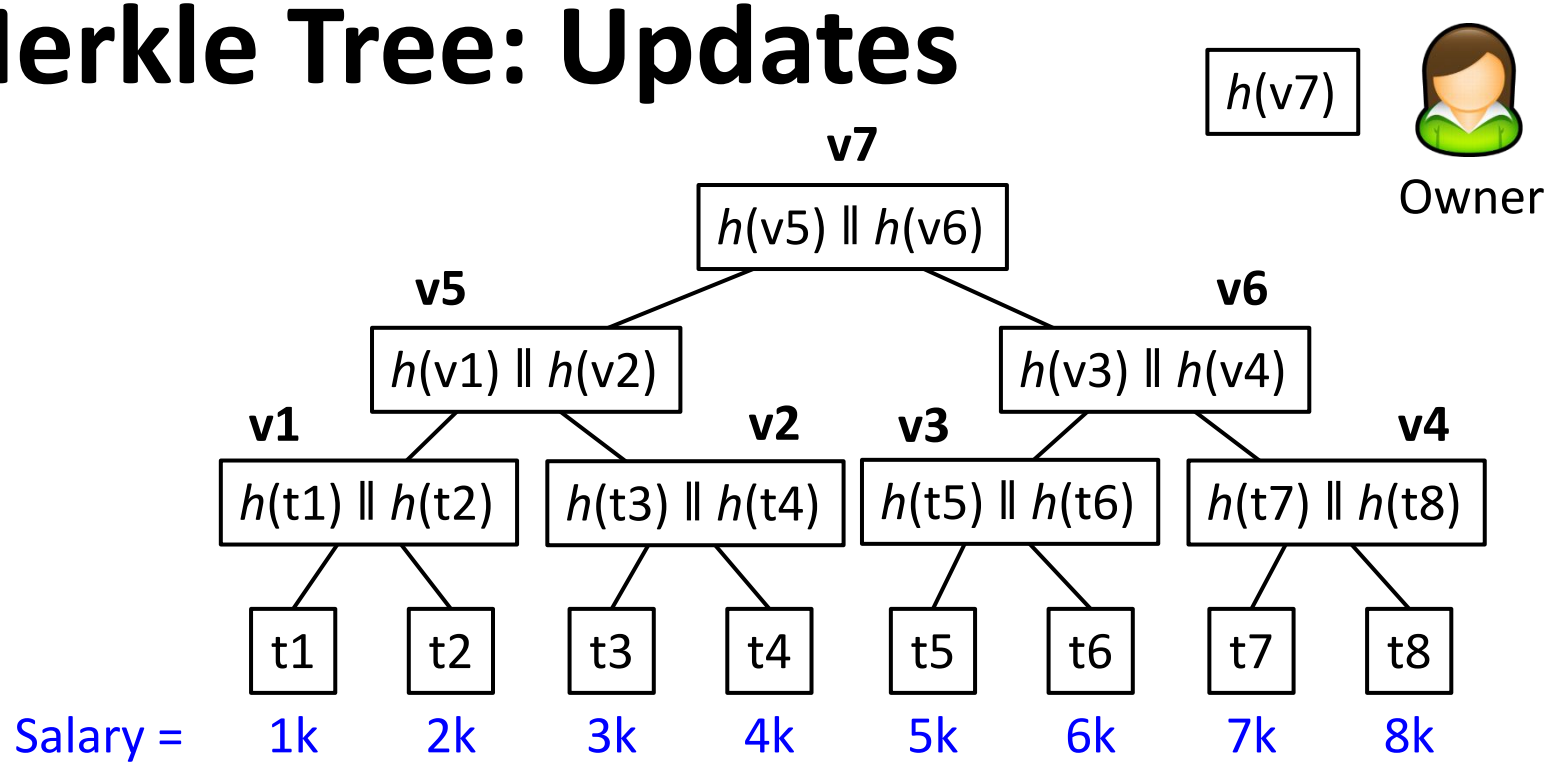
- Consider a query on "Salary = 4k"
- What should the service provider return?

Merkle Tree: Exercise



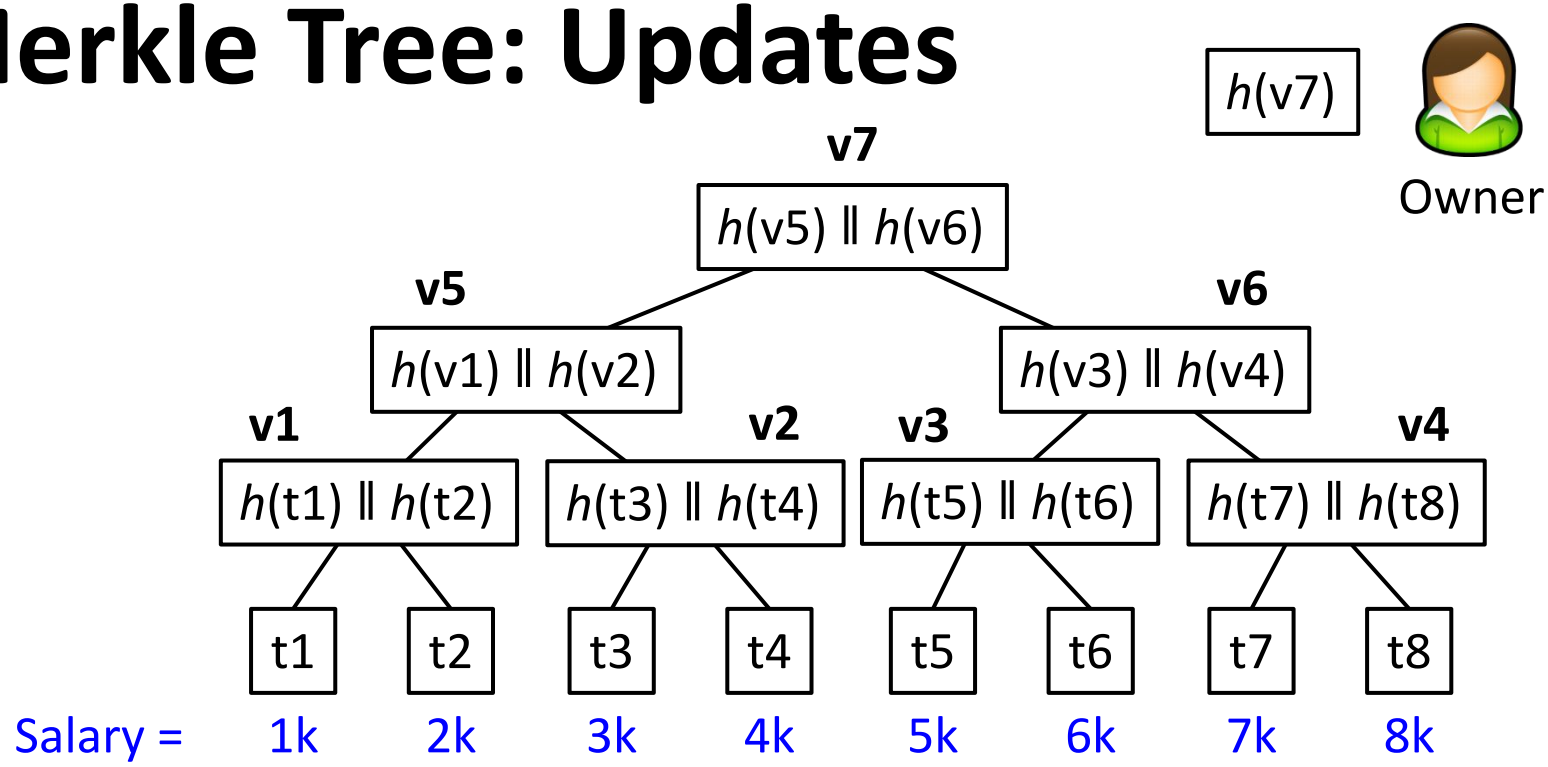
- Consider a query on "Salary in (3k, 4k]"
- What should the service provider return?

Merkle Tree: Updates



- What if the tuples are updated?
  - E.g., the tuple with Salary = 6k is deleted
- Option 1:
  - The data owner reconstructs the binary tree, re-computes *h*(root), signs it with a timestamp, and sends it to the service provider
  - The data owner announces the timestamp to users

Merkle Tree: Updates



- Problem with Option 1:
  - Reconstructing the whole binary tree once per update is time-consuming
- Improved solution:
  - Use an update-friendly tree structure, e.g., a red-black tree
  - Any tree structure could be signed by the data owner like a Merkle tree

Extension to Multi-Dimensional Data

- The previous discussions focus on one-dimensional queries
- How about multi-dimensional queries? e.g.,
  - SELECT * FROM Employee
    WHERE Age > 30 AND Salary > 10000
- Option 1:
  - Build two Merkle trees on Age and Salary, respectively
- Problem:
  - No benefit from using two trees simultaneously
  - We need to choose either the tree on Age or the one on Salary to answer a query

Extension to Multi-Dimensional Data

- The previous discussions focus on one-dimensional queries
- How about multi-dimensional queries? e.g.,
  - SELECT * FROM Employee
    WHERE Age > 30 AND Salary > 10000
- Option 2:
  - Use multi-dimensional indices
  - E.g., an R-tree

R-tree

name	semester	credits
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50